

UNIVERSIDADE FEDERAL DE MATO GROSSO
INSTITUTO DE FÍSICA
PROGRAMA DE PÓS GRADUAÇÃO EM FÍSICA AMBIENTAL

Adaptação de algoritmos de processamento de
dados ambientais para o contexto de *Big Data*

Guilherme Falcão da Silva Campos

Orientador: Prof. Dr. Josiel Maimone de Figueiredo

Cuiabá - MT
Julho/2015

UNIVERSIDADE FEDERAL DE MATO GROSSO
INSTITUTO DE FÍSICA
PROGRAMA DE PÓS GRADUAÇÃO EM FÍSICA AMBIENTAL

Adaptação de algoritmos de processamento de
dados ambientais para o contexto de *Big Data*

Guilherme Falcão da Silva Campos

Dissertação apresentada ao Programa de Pós Graduação em Física Ambiental da Universidade Federal de Mato Grosso, como parte dos requisitos para obtenção do título de Mestre em Física Ambiental.

Prof. Dr. Josiel Maimone de Figueiredo

Cuiabá, MT

Julho/2015

Dados Internacionais de Catalogação na Fonte.

C198a Campos, Guilherme Falcão da Silva.
Adaptação de algoritmos de processamento de dados ambientais para o contexto de Big Data / Guilherme Falcão da Silva Campos. -- 2015
57 f. ; 30 cm.

Orientador: Josiel Maimone de Figueiredo.
Dissertação (mestrado) - Universidade Federal de Mato Grosso, Instituto de Física, Programa de Pós-Graduação em Física Ambiental, Cuiabá, 2015.
Inclui bibliografia.

1. MapReduce. 2. Hadoop. 3. Wavelets. 4. Dimensão Fractal. 5. Dados Ambientais. I. Título.

Ficha catalográfica elaborada automaticamente de acordo com os dados fornecidos pelo(a) autor(a).

Permitida a reprodução parcial ou total, desde que citada a fonte.

UNIVERSIDADE FEDERAL DE MATO GROSSO
INSTITUTO DE FÍSICA
Programa de Pós-Graduação em Física Ambiental

FOLHA DE APROVAÇÃO

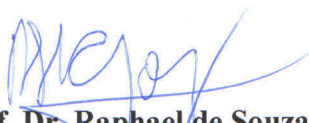
TÍTULO: ADAPTAÇÃO DE ALGORITMOS DE PROCESSAMENTO
DE DADOS AMBIENTAIS PARA O CONTEXTO DE *BIG*
DATA

AUTOR: GUILHERME FALCÃO DA SILVA CAMPOS

Dissertação de Mestrado defendida e aprovada em 23 de novembro de 2015, pela
comissão julgadora:



Prof. Dr. Josiel Maimone de Figueiredo
Orientador
Instituto de Computação – UFMT



Prof. Dr. Raphael de Souza Rosa Gomes
Examinador Interno
Instituto de Computação – UFMT



Prof. Dr. Marcos Rodrigues Vieira
Examinador Externo
Instituto de Ciências Matemáticas e de Computação - ICMC/USP

DEDICATÓRIA

*A Deus, aos meus pais, Pio e Angela, à
minha amada esposa Juliana e à minha
irmã Isabella.*

Agradecimentos

- A Deus antes de tudo, por ter me dado a vida e a capacidade de realizar este trabalho.
- Aos meus pais, pelo apoio e dedicação em todos os momentos da minha vida.
- A minha esposa Juliana que nunca me deixou desistir e nem desanimar e sempre foi compreensiva nos momentos mais difíceis.
- Ao meu orientador Josiel, pelos puxões de orelha e também pela paciência.
- Aos meus colegas de mestrado que me auxiliaram nas disciplinas.
- Ao Raphael Souza, pelo auxílio na compreensão dos algoritmos.
- Ao Cesar, pelo auxílio na área de infraestrutura e à Gracyeli pelo auxílio com os algoritmos.
- À Priscilla Ramsay, pela revisão ortográfica.
- Aos amigos Alvaro, André e Madson, que, querendo ou não, fizeram parte deste trabalho.
- Ao Rodicrisller, pelas caronas às reuniões.
- Ao CNPq pelo auxílio financeiro.
- Por fim, a todos que de alguma maneira, direta ou indiretamente, contribuíram para o sucesso deste trabalho.

*Os céus declaram a glória de Deus e
o firmamento anuncia a obra das suas
mãos.*

Bíblia Sagrada - Salmos 19:1.

SUMÁRIO

LISTA DE FIGURAS	II
LISTA DE SIGLAS	III
RESUMO	IV
ABSTRACT	V
1 INTRODUÇÃO	1
1.1 OBJETIVO GERAL	2
1.2 OBJETIVOS ESPECÍFICOS	2
2 REVISÃO BIBLIOGRÁFICA	3
2.1 SÉRIES TEMPORAIS	3
2.1.1 Dados Ambientais	3
2.1.2 Análise de Séries Temporais	4
2.1.3 Transformada de <i>Wavelet</i>	4
2.1.4 Busca por Similaridade em Séries Temporais	6
2.1.5 Análise de Séries Temporais caóticas	7
2.1.5.1 Reconstrução do espaço de fase	8
2.1.5.2 Dimensão Fractal	9
2.1.5.2.1 Dimensão Box-Counting	10
2.1.5.2.2 Dimensão de Correlação	10
2.2 BIG DATA	12
2.3 HADOOP	13
2.3.1 Hadoop Distributed File System (HDFS)	14
2.3.1.1 Arquitetura do HDFS	15
2.3.2 MapReduce	18

3	MATERIAIS E MÉTODOS	23
3.1	AMBIENTE DE DESENVOLVIMENTO	23
3.2	DADOS UTILIZADOS	24
3.2.1	Busca por similaridade	24
3.2.2	Fractais	24
3.2.2.1	Dimensão <i>box-counting</i>	25
3.2.2.2	Dimensão de correlação	26
3.3	ADAPTAÇÃO DOS ALGORITMOS	27
3.3.1	<i>kNN Search</i>	27
3.3.2	Transformada de <i>Wavelet</i>	27
3.3.3	Dimensão Fractal	28
3.3.3.1	Dimensão <i>Box-Counting</i>	28
3.3.3.2	Dimensão de Correlação	29
4	RESULTADOS E DISCUSSÃO	30
4.1	IMPLEMENTAÇÃO DA APLICAÇÃO	30
4.2	BUSCA POR SIMILARIDADE EM SÉRIES TEMPORAIS . . .	31
4.2.1	Implementação	31
4.2.2	Tratamento dos Dados	31
4.2.3	Transformada de <i>Wavelet</i>	32
4.2.4	Criação das <i>wavelets</i>	33
4.2.5	<i>kNN Search</i>	36
4.2.6	Busca em <i>wavelets</i>	37
4.3	SÉRIES TEMPORAIS CAÓTICAS	39
4.3.1	Dimensão Box-Counting	39
4.3.1.1	Implementação	40
4.3.1.2	Teste de escalabilidade - Retas	43
4.3.1.3	Dimensões calculadas	45
4.3.2	Dimensão de Correlação	47
4.3.2.1	Tratamento dos Dados	47
4.3.2.2	Implementação	48

5	CONSIDERAÇÕES FINAIS	52
5.1	CONTRIBUIÇÕES	53
5.2	TRABALHOS FUTUROS	53
Apêndice A: Implementação Transformada de Wavelet		1
A.1	Mapper	1
A.2	Reducer	3
Apêndice B: Implementação kNN Search		1
B.1	Mapper	1
B.2	Combiner	3
B.3	Reducer	5
Apêndice C: Implementação Box-Counting		1
C.1	Mapper - Job 1	1
C.2	Combiner - Job 1	2
C.3	Reducer - Job 1	2
C.4	Mapper - Job 2	3
C.5	Combiner - Job 2	4
C.6	Reducer - Job 2	5
Apêndice D: Implementação Dimensão de Correlação		1
D.1	Mapper	1
D.2	Reducer	4

LISTA DE FIGURAS

1	Parte real da <i>wavelet</i> de Morlet.	5
2	Multiresolução dada pela transformada de <i>wavelet</i> (Bolzan, 2006)	6
3	Atrator de Lorenz	8
4	Atrator de Lorenz reconstruído em duas dimensões, utilizando a técnica de defasagem temporal.	9
5	$\log(C(r)) \times \log(r)$ para valores crescentes de m para um atrator reconstruído a partir da série X de Lorenz.	11
6	Um cliente HDFS criando um novo arquivo (Shvachko et al., 2010).	17
7	Arquitetura HDFS (Wang et al., 2013).	18
8	MapReduce no Hadoop. Adaptado de Wang et al. (2013).	21
9	Fluxo de dados no <i>Hadoop</i> (Wang et al., 2013).	22
10	Hénon Map	26
11	Curva de Koch	26
12	Estrutura da aplicação desenvolvida.	30
13	Diagrama de Atividade - Transformada de <i>Wavelet</i>	33
14	Tamanho dos arquivos criados e a quantidade de registros arma- zenados.	34
15	Tamanho médio de cada <i>wavelet</i>	35
16	Tempo de cada fase e tempo total comparados.	35
17	Comparação do tempo sequencial com o tempo do <i>Hadoop</i>	36
18	Diagrama de Atividade - <i>kNN Search</i>	37
19	Tempo de cada fase e tempo total comparados.	38
20	Comparação do tempo do sequencial com o tempo do <i>Hadoop</i>	39
21	Diagrama de Atividade - <i>Box-Counting</i>	40
22	Tamanho dos arquivos intermediários gerados pelo primeiro <i>job</i> para diferentes quantidades de pontos.	42

23	Tempos de cada <i>job</i> comparados.	42
24	Tempos de execução para arquivos diferentes.	43
25	Regressões criadas para as execuções sequenciais.	44
26	Tempos de execução para o <i>hadoop</i> e os sequenciais estimados. . .	44
27	Dimensão <i>Box-Counting</i> calculada para uma reta.	45
28	Dimensão <i>Box-Counting</i> calculada para um retângulo.	46
29	Dimensão <i>Box-Counting</i> calculada para o mapa de Hénon.	46
30	Dimensão <i>Box-Counting</i> calculada para a curva de Koch.	47
31	Diagrama de Atividade - Dimensão de Correlação	49
32	Tempo de execução para diferentes quantidades de pontos.	50
33	<i>Zoom</i> na parte inicial da Figura 32.	51

LISTA DE SIGLAS

GFS : *Google File System*

HDFS : *Hadoop Distributed Filesystem*

kNN : *k-Nearest Neighbors*

CSV : *Comma-Separated Values*

CUDA : *Compute Unified Device Architecture*

FPGA : *Field-Programmable Gate Array*

RESUMO

CAMPOS, G. F. S. Adaptação de algoritmos de processamento de dados ambientais para o contexto de *Big Data*, 2015, 6f. Dissertação (Mestrado em Física Ambiental) - Instituto de Física, Universidade Federal de Mato Grosso.

Pesquisas ambientais dependem de dados de sensores para a criação das séries temporais referentes às variáveis analisadas. A quantidade de dados tende a aumentar, cada vez mais, à medida que novos sensores são criados e instalados. Com o passar do tempo os conjuntos de dados se tornam massivos, requerendo novas formas de armazenamento e processamento. Este trabalho busca meios de se contornar esses problemas utilizando uma solução tecnológica capaz de armazenar e processar grandes quantidades de dados. A solução tecnológica utilizada é o *Apache Hadoop*, uma ferramenta voltada a problemas de *Big Data*. Com a finalidade de avaliar a ferramenta foram utilizados diferentes conjuntos de dados e adaptados diferentes algoritmos usados na análise de séries temporais. Foram implementados análises de séries caóticas e não caóticas. As implementações foram a transformada de *wavelet*, uma busca por similaridade usando a função de distância Euclidiana, cálculo da dimensão *box-counting* e o cálculo da dimensão de correlação. Essas implementações foram adaptadas para utilizar o paradigma de processamento distribuído *MapReduce*.

Palavras-Chave: MapReduce, Hadoop, wavelets, dimensão fractal, dados ambientais.

ABSTRACT

CAMPOS, G. F. S. Adaptation of environmental data processing algorithms to the *Big Data* context, 2015, 6f. Dissertação (Mestrado em Física Ambiental) - Instituto de Física, Universidade Federal de Mato Grosso.

Environmental research depend on sensor generated data to create time series regarding the variables that are being analyzed. The amount of data tends to increase as more and more sensors are created and installed. After some time the datasets become huge and requires new ways to process and store the data. This work seeks to find ways to avoid these issues using a technological solution able to store and process large amounts of data. The solution used is Apache Hadoop, a tool which purpose is to solve Big Data problems. In order to evaluate the tool were used different datasets and time series analysis algorithms. The analysis of chaotic and non-chaotic time series were implemented. These implementations were: the wavelet transform, similarity search using Euclidean distance function, the calculus of the box-counting dimension and the calculus of the correlation dimension. Those implementations were adapted for the MapReduce parallel processing paradigm.

Keywords: MapReduce, Hadoop, wavelets, fractal dimension, environmental data.

1 INTRODUÇÃO

A explosão na quantidade de informações geradas é uma realidade. A aquisição de dados de diversas fontes busca compreender, prever e estudar diversos fenômenos que cercam a sociedade, em especial fenômenos meteorológicos, pois estes influenciam diretamente o comportamento da sociedade.

O estudo de fenômenos meteorológicos depende da extração de dados que demonstrem o estado atual do meio ambiente, como temperatura, precipitação, umidade relativa do ar, radiação solar, fluxo de calor, entre outros. O estudo dessas variáveis busca entender o comportamento e a dinâmica do ambiente em que a sociedade está inserida.

Os dados utilizados nas pesquisas ambientais tem suas origens em diversos equipamentos. Existem sensores pontuais instalados em torres meteorológicas localizadas em diversos locais: centros urbanos, áreas alagadas, florestas, cerrado, desertos, entre outros. E existem outras fontes de dados nos satélites que estão em órbita colhendo informações de diversos locais ao redor do planeta.

Independente de onde os dados foram obtidos é necessária sua análise, que busca entender os fenômenos que cercam a sociedade. Esses dados possuem várias formas, algumas mais simples, outras mais complexas, e podem compor *terabytes* de informação. Esses dados, ao serem analisados, geram outros dados que vão auxiliar o processo de tomada de decisão da sociedade.

Essa quantidade de dados volumosa e complexa é difícil de ser tratada, armazenada, analisada e consultada por meios convencionais. Os conjuntos de dados podem superar a capacidade de armazenamento de um único computador, fazendo-se necessária a utilização de várias máquinas interligadas, um sistema distribuído. A computação distribuída permite que diversos computadores interligados trabalhem de forma que possam prover armazenamento e processamento de dados.

Este trabalho busca utilizar uma ferramenta distribuída chamada *Ha-*

doop, que cria um ambiente que suporta grandes quantidades de dados em múltiplos computadores interligados, para lidar com as dificuldades de armazenamento de grandes quantidades de dados ambientais, seu tratamento e análise de suas estruturas complexas.

1.1 OBJETIVO GERAL

O objetivo deste trabalho é adaptar métodos de estudos e análises de séries temporais ao ambiente de computação distribuída, visando melhorar o modo de armazenamento, tratamento e processamento desses conjuntos crescentes de dados complexos.

1.2 OBJETIVOS ESPECÍFICOS

- Estudar análise de séries temporais ambientais.
- Estudar análises de séries não lineares.
- Estudar uma ferramenta distribuída
- Adaptar as análises na ferramenta distribuída.
- Testar as adaptações.
- Avaliar as adaptações.

O próximo capítulo trata da fundamentação teórica utilizada neste trabalho, os diferentes modos de análises de diferentes séries temporais e o conceito de *big data* que busca resolver o problema do grande volume de dados.

2 REVISÃO BIBLIOGRÁFICA

Neste capítulo é abordado o modo com que dados ambientais são representados como séries temporais, como as séries temporais caóticas podem ser analisadas e estudadas e como a utilização de ferramentas de *Big Data* podem auxiliar na compreensão, extração de conhecimento e funcionamento das séries estudadas.

2.1 SÉRIES TEMPORAIS

Um conjunto de dados obtidos a partir de observações feitas sequencialmente ao longo do tempo é denominada de *série temporal*. Ela pode consistir em valores observados em intervalos discretos, uma média em um intervalo de tempo ou em valores gravados constantemente ao longo do tempo (Machiwal e Jha, 2012). De modo geral, uma série temporal de n valores observados é definida como um vetor $Z(t) = [Z(t_1), Z(t_2), \dots, Z(t_n)]$, onde $t_1, t_2, \dots, t_n$ são os instantes observados (Morettin e Toloí, 1981).

Dados nas áreas econômicas, negócios, engenharia, meio-ambiente, medicina, ciências da terra, hidrologia, meteorologia, climatologia e outras áreas, são normalmente coletados na forma de série temporal. Alguns exemplos são valores das ações em dias consecutivos, a medida da temperatura do ar feita em minutos/horas/dias/meses/anos (Machiwal e Jha, 2012).

2.1.1 Dados Ambientais

A obtenção de dados ambientais atualmente é feita de modo automatizado e por meio de sensores, esses dados são guardados na forma de séries temporais. A utilização de sensores faz com que a quantidade de dados gerados e manipulados nos estudos cheguem a casa de *terabytes*. No contexto da física ambiental são medidos dados micrometeorológicos através de vários sensores em

intervalos de segundos. Cada variável medida é uma série temporal que será analisada utilizando métodos de análises de séries temporais (Gomes, 2012; Fang et al., 2014).

2.1.2 Análise de Séries Temporais

A análise de séries temporais é a investigação de uma sequência de dados distribuídos em um intervalo de tempo, onde o tempo é uma variável independente. Essa análise por ser uma busca pelo mecanismo gerador da série, a criação de um modelo de previsão de valores futuros, descrição do comportamento da série e a busca por periodicidades (Morettin e Toloi, 1981).

Para a análise de séries temporais foi utilizada a transformada de *wavelet* e a busca por similaridade, a análise de séries temporais caóticas foi feita utilizando o cálculo da dimensão fractal e a reconstrução do espaço de fase a partir de uma série temporal. Esses conceitos são apresentados nas próximas seções.

2.1.3 Transformada de *Wavelet*

No estudo de séries temporais a transformada de *wavelet* permite analisar a periodicidade de eventos em diferentes escalas de variabilidade temporal sem necessitar de uma série estacionária. Assim a ferramenta permite analisar eventos irregulares distribuídos em diferentes frequências (Santos et al., 2013). A análise de *wavelet* está tornando-se uma ferramenta comum na busca dos picos em séries temporais. Ao decompor a série no domínio de tempo-frequência é possível observar o comportamento dominante na série e como esse comportamento varia com o tempo (Torrence e Compo, 1998).

A transformada de *wavelet* está sendo usada em diversas áreas de conhecimento, incluindo processamento de imagens, processamento de sinais, geofísica, sistemas hidrológicos, sistemas climáticos, entre outras (Ong, 2010; Torrence e Webster, 1999; Darilmaz, 2006; Bolzan, 2006).

A transformação de uma série temporal para o domínio de frequências pode ser feito através de outras ferramentas matemáticas. A mais conhecida

delas é a transformada de Fourier, desenvolvida em 1807 por Joseph Fourier. Ela se baseia na afirmação que toda função real periódica pode ser escrita por uma soma de senos e cossenos. Mas a maioria das séries temporais apresentam uma variabilidade aperiódica, pois são séries não-estacionárias, dificultando assim sua análise a partir da transformada de Fourier (Santos et al., 2013).

Nos casos de séries não-estacionárias, Gabor, em 1946, dividiu a série em segmentos fixos e aplicou a transformada de Fourier nesses segmentos. Esse método ficou conhecido como Transformada de Fourier Janelada, que é de grande contribuição para o estudo de séries não-estacionárias, mas ainda não resolvia o problema de que a janela era fixa e que as funções trigonométricas possuem energia infinita (Bolzan, 2006; Santos et al., 2013).

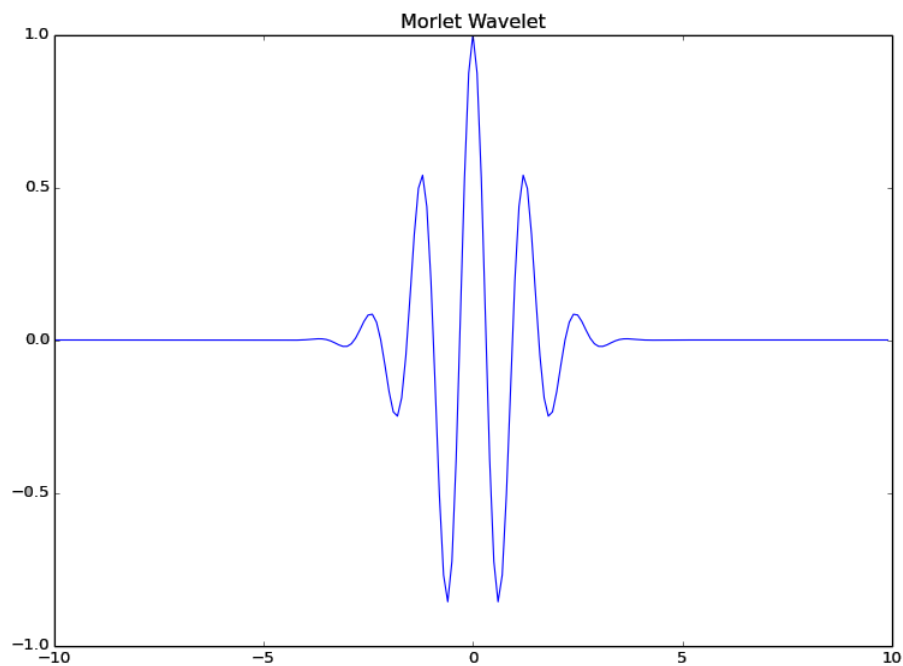


Figura 1: Parte real da *wavelet* de Morlet.

A análise de *wavelet* busca resolver esses problemas ao aplicar um "pacote" de onda com duração finita e com uma frequência específica. Esse formato é visto na Figura 1 e pode ser usado como uma janela no estudo da variação nas séries temporais. Ele possui a vantagem de ter um período determinado e extensão finita. Então, ao deslizar a *wavelet* pela série, obtém-se o valor da potência dessa *wavelet* em vários pontos da série, gerando uma nova série de amplitude

e tempo. Também a escala da *wavelet* é alterada fazendo com que regiões de frequências diferentes sejam identificadas (Santos et al., 2013).

Então a transformada de *wavelet* é obtida a partir de dilatações e translação de uma função base $\psi(t)$, chamada de *wavelet*-mãe. As *wavelets* são geradas por dilatações $\psi(t) \rightarrow \psi(2t)$ e translações $\psi(t) \rightarrow \psi(t + 1)$ da função base geradora $\psi(t)$. Então as wavelets-filhas são geradas a partir da *wavelet*-mãe de acordo com a Equação 1, onde j e k são dilatações e translações, respectivamente.

$$\psi_{j,k}(t) = \frac{1}{\sqrt{2}} \psi \left(\frac{t - k}{j} \right) \quad (1)$$

Assim a transformada é definida como:

$$\langle f, \psi_{j,k} \rangle = \int_{-\infty}^{\infty} f(t) \psi \left(\frac{t - k}{j} \right) dt \quad (2)$$

A ideia central da transformada de *wavelet* consiste em decompor uma série em diferentes resoluções. Essas resoluções caracterizam diferentes estruturas nas séries, facilitando a interpretação de informação. A Figura 2 mostra esquematicamente esse tipo de análise (Bolzan, 2006).

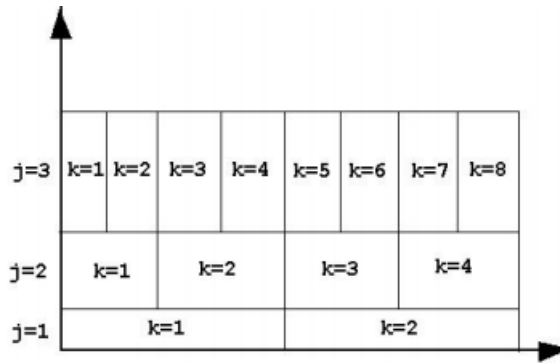


Figura 2: Multiresolução dada pela transformada de *wavelet* (Bolzan, 2006)

2.1.4 Busca por Similaridade em Séries Temporais

As séries temporais possuem diversos comportamentos. Para se analisar os diferentes comportamentos é necessário que uma série possa ser comparada com outra série. Para isso foram desenvolvidas diversas funções de distâncias que

buscam dizer o quão similares as séries comparadas são. Essas funções buscam aproximar as séries que possuem as mesmas características.

A função de distância mais utilizada na medida de similaridade entre duas séries temporais é a distância euclidiana $D(\vec{x}, \vec{y})$, onde $\vec{x}$ e $\vec{y}$ são séries temporais. A distância é dada por:

$$D(\vec{x}, \vec{y}) = \left(\sum_{i=0}^{n-1} |x_i - y_i|^2 \right)^{\frac{1}{2}} \quad (3)$$

Utilizando esse modelo é possível comparar diversas séries temporais e saber quais são mais próximas. A busca pelas séries mais próximas é denominada *kNN Search*, que é a busca pelos k vizinhos mais próximos de uma determinada série (Chan e Fu, 1999).

2.1.5 Análise de Séries Temporais caóticas

A análise de séries temporais é a investigação dos dados temporais de modo que seja possível o entendimento do comportamento da série, suas características e seu comportamento ao longo do tempo (Machiwal e Jha, 2012). No caso de séries caóticas são utilizadas técnicas como a reconstrução do espaço de fase afim de se identificar o atrator da série. Essa reconstrução é feita utilizando-se o método da defasagem temporal (Takens, 1981). O espaço de fase é um espaço matemático definido por coordenadas ortogonais que são variáveis que definem um estado instantâneo do sistema, assim a passagem do tempo fica implícita (Baker, 1996).

O atrator representa a tendência das trajetórias. Um exemplo de atrator é o que foi construído a partir das equações 4 propostas por Lorenz (Lorenz, 1963) e é demonstrado na figura 3.

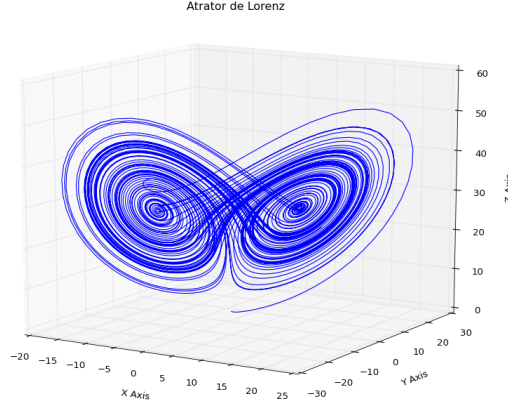


Figura 3: Atrator de Lorenz

2.1.5.1 Reconstrução do espaço de fase

Em sistemas experimentais onde as equações dinâmicas não são conhecidas se recorre à reconstrução do atrator. É possível observar a dinâmica do sistema através de uma única série temporal (Baker, 1996). Esse conceito foi demonstrado por Takens (1981).

A reconstrução do espaço de fase do atrator a partir de uma única série temporal requer a geração de variáveis adicionais, que são criadas com a utilização da defasagem temporal. Temos como a série inicial um vetor $X_0(t)$ criando novas variáveis $X_k(t)$, onde k inicia em 1 vai até n . A defasagem fixa τ é aplicada:

$$X_0 = \{X_0(t_1), \dots, X_0(t_n)\}$$

$$X_1 = \{X_0(t_1 + \tau), \dots, X_0(t_n + \tau)\}$$

$$X_2 = \{X_0(t_1 + 2\tau), \dots, X_0(t_n + 2\tau)\}$$

...

$$X_{n-1} = \{X_0(t_1 + (n-1)\tau), \dots, X_0(t_n + (n-1)\tau)\}$$

A qualidade do atrator reconstruído é sensível ao τ escolhido, então é necessário escolher um que contenha o maior ganho de informação disponível.

O atrator de Lorenz mostrado na figura 3 é modelado pelo sistema dinâmico

mico a seguir:

$$\begin{cases} \partial x / \partial t = -\alpha(x + y) \\ \partial y / \partial t = -xy + rx - y \\ \partial z / \partial t = xy - bz \end{cases} \quad (4)$$

A figura 4 mostra o espaço de fase do atrator reconstruído com a utilização de defasagem temporal. Foi utilizado como série temporal $X(t)$ e construída uma outra série $X(t + \tau)$. Pode se observar que ambas as figuras mantêm a aparência da "borboleta" (Baker, 1996).

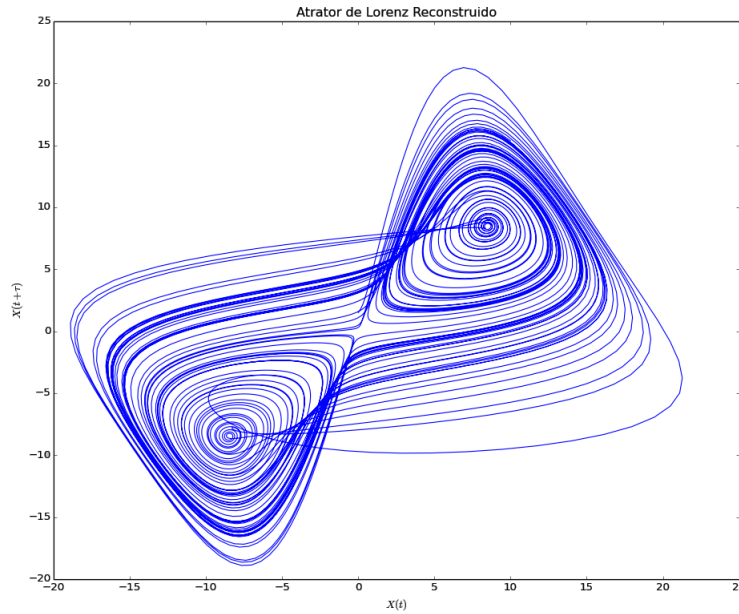


Figura 4: Atrator de Lorenz reconstruído em duas dimensões, utilizando a técnica de defasagem temporal.

2.1.5.2 Dimensão Fractal

No estudo das estruturas geradas dos espaços de fases complexos, pode-se perceber que o objeto apresenta as mesmas características para diferentes variações de escala e tamanho, portanto ele apresenta auto-similaridade, que é uma característica dos fractais. Na teoria dos fractais, o conceito de dimensão fractal é muito importante pois provê uma estimativa da dimensionalidade intrínseca da série estudada (Nunes et al., 2010). Assim, a dimensão fractal é uma medida da densidade da série no espaço de fase definido. Na literatura existem diver-

sas dimensões fractais, uma das mais usada é a *dimensão de correlação* que foi definida por Grassberger e Procaccia (1983). Outra muito usada é a dimensão *Box-Counting* (Baker, 1996; Nunes et al., 2010).

2.1.5.2.1 Dimensão Box-Counting -

A dimensão *Box-Counting*, ou contagem de caixas, é estimada a partir do seguinte método. Considere uma linha uni-dimensional. Ela pode ser coberta por $N(\varepsilon)$ caixas uni-dimensionais de tamanho ε . Se L é o tamanho da linha, então $N(\varepsilon) = L (1/\varepsilon)$. Do mesmo modo, um quadrado bi-dimensional de lado L pode ser coberto por $N(\varepsilon) = L^2 (1/\varepsilon)^2$. Para um cubo de tri-dimensional o expoente seria 3, e assim sucessivamente para dimensões mais altas. Generalizando,

$$N(\varepsilon) = L^d (1/\varepsilon)^d. \quad (5)$$

Utilizando logaritmos obtém-se

$$d = \frac{\log N(\varepsilon)}{\log L + \log (1/\varepsilon)} \quad (6)$$

e no limite de um ε pequeno o termo que envolve L se torna negligenciável. A dimensão *box-counting* é definida como

$$d_C = \lim_{\varepsilon \rightarrow 0} \frac{\log N(\varepsilon)}{\log (1/\varepsilon)} \quad (7)$$

Uma abordagem equivalente é considerar d_C a inclinação da curva formada por $\log N$ x $\log(1/\varepsilon)$ enquanto $\varepsilon \rightarrow 0$ (Baker, 1996).

2.1.5.2.2 Dimensão de Correlação -

A dimensão de correlação foi definida por Grassberger e Procaccia (1983). Em um espaço m -dimensional, a função de correlação integrante $C(r)$ é mostrada na equação 8:

$$C(r) = \frac{1}{n^2} \sum_{\substack{i,j=1 \\ i \neq j}} \theta(r - |X_i - X_j|) \quad (8)$$

Onde X_i e X_j são pontos no atrator, N é o número de pontos selecionados na série e θ é a função *Heavyside* definida na equação 9. A função *Heavyside* conta o número de pontos dentro de uma esfera de raio r com centro em X_i , e $C(r)$ é uma fração média dos pontos dentro de r .

$$\theta(x) = \begin{cases} 0, & \text{se } x < 0 \\ 1, & \text{se } x \geq 0 \end{cases} \quad (9)$$

Desta forma, a dimensão de correlação é definida como a variação de $C(r)$ com r :

$$C(r) \approx r^{d_G} \text{ com } r \rightarrow 0 \quad (10)$$

Então pode-se obter o valor da dimensão de correlação através da inclinação da reta $\log(C(r)) \times \log(r)$. A figura 5 mostra o gráfico para um atrator reconstruído em m dimensões a partir da série X do atrator de Lorenz. A parte linear tem uma inclinação d_G (Baker, 1996).

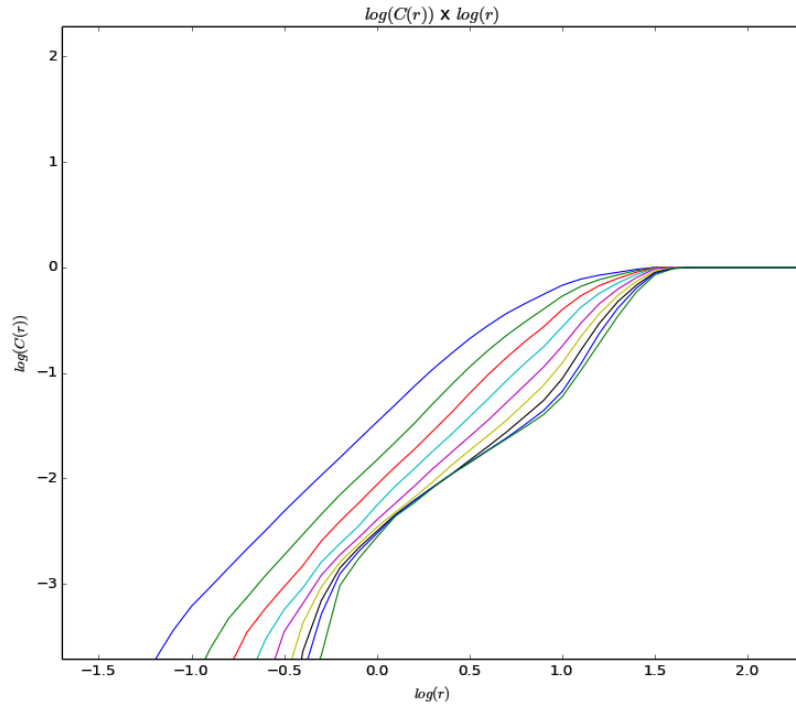


Figura 5: $\log(C(r)) \times \log(r)$ para valores crescentes de m para um atrator reconstruído a partir da série X de Lorenz.

A próxima seção trata de uma abordagem computacional que advém da necessidade de análise, armazenamento e tratamento de grandes volumes de dados.

2.2 BIG DATA

Big Data é um conceito que se refere a uma enorme quantidade de dados não-estruturados gerados por diferentes tipos de aplicações e como é possível o manuseio e interpretação desses dados. Eles podem ser gerados em diversos domínios de aplicação como, por exemplo, aplicações científicas, redes sociais, *logs* da Web, governo eletrônico, sistemas médicos, redes de sensores, dados de sensoramento remoto, entre outros. Os dados armazenados nesses cenários têm em comum o seu grande volume, a dificuldade em serem analisados e diferentes formas de armazenamento, desde dados brutos de sensores a imagens de satélite. (Cuzzocrea et al., 2011).

Eles são tão volumosos e complexos que a utilização de SGBDs tradicionais é ineficiente para tratá-los, armazená-los e consultá-los. Os dados necessitam de execução de consultas com baixa latência, do tratamento dos grandes volumes de dados com escalabilidade elástica horizontal, do suporte a modelos flexíveis de armazenamento e suporte simples à replicação de distribuição de dados. Para resolver e atender as novas demandas, ferramentas voltadas para os problemas de *Big Data* foram criadas. Uma dessas ferramentas é o Apache Hadoop que é mostrado na seção 2.3 (Vieira et al., 2012).

Observando o domínio de atuação do *Big Data* é visível a existência de três características principais que podem ser usadas para identificar ou definir um problema de *Big Data*: Volume, Variedade e Velocidade. Volume se refere ao tamanho do conjunto de dados, como *terabytes*, *petabytes*, etc. Variedade se refere aos tipos de dados, que podem ser originados de diversas fontes e ser não-estruturados, como imagens, sensores, sinais de GPS e muitos outros. Finalmente, Velocidade se refere ao intervalo de tempo nos quais os dados são gerados, como horas, minutos, segundos, e também o tempo necessário para se processar os

dados, onde dependendo da necessidade, podem ser processados em tempo real. (Zaslavsky et al., 2013; McAfee e Brynjolfsson, 2012).

As três características principais do *Big Data* podem ser observados nos estudos ambientais e meteorológicos. A captação de informações através dos diferentes sensores e criação das séries temporais tem o potencial de gerar uma grande quantidade de dados em pouco tempo. Isso representa uma dificuldade na hora de armazenar e processar esses dados de forma eficiente (Zaslavsky et al., 2013).

A próxima seção trata de uma ferramenta específica no contexto de *Big Data*, o Apache Hadoop, um projeto *open source* que permite armazenar e analisar grandes quantidades e variedades de dados.

2.3 HADOOP

O Hadoop foi criado por Doug Cutting e teve sua origem no projeto Apache Nutch, um projeto que visava criar um *web crawler* e indexador de páginas Web *open source*. No início do projeto, ao criarem o primeiro *crawler*, foi percebido que a arquitetura original não comportaria bilhões de páginas Web. A solução veio com a publicação do artigo de Ghemawat et al. (2003) que especificava a arquitetura do sistema de arquivo distribuído do Google (GFS). Algo como o GFS solucionaria os problemas de armazenamento de grandes quantidades de dados do projeto Nutch. Então eles implementaram uma versão *open source* do GFS, o *Nutch Distributed Filesystem* (NDFS).

Posteriormente, o Google publicou o artigo de Dean e Ghemawat (2008) que definiu um modelo de programação para processamento de grandes conjuntos de dados chamado de *MapReduce*. A equipe do projeto Nutch uniu as duas coisas, o NDFS e a implementação do *MapReduce*, e os moveram para fora do projeto, criando um novo projeto independente chamado Hadoop. Hoje, em seu cerne, o Hadoop provê um sistema de armazenamento distribuído com tolerância a falhas e um modo de acessar e analisar os dados armazenados.

O Hadoop serve de complemento para as tecnologias já existentes, pois

ele possibilita a análise de um conjunto extenso de dados através do *MapReduce*, trabalha com dados não estruturados e é melhor aplicado onde os dados são escritos apenas uma vez e consultados várias vezes, como o caso dos sensores ambientais.

A seguir será mostrado em maiores detalhes o funcionamento e arquitetura do Hadoop: o funcionamento do HDFS e a implementação e funcionamento do *MapReduce*.

2.3.1 Hadoop Distributed File System (HDFS)

O HDFS é o sistema de arquivos distribuído do Hadoop. Ele separa o armazenamento dos dados das aplicações e dos metadados, do mesmo modo que o GFS faz. Os metadados ficam armazenados em um servidor dedicado chamado de *NameNode*. Já os dados de aplicações ficam armazenados em outros servidores, os *DataNode*. Os servidores são interconectados utilizando protocolos baseados no TCP.

É necessário que o *NameNode* tenha o seu conteúdo replicado em outro servidor para se obter confiabilidade, pois sem o *NameNode* todo o sistema de arquivos não pode ser usado. Já com os *DataNodes*, não é necessária essa preocupação, pois a replicação de dados de aplicação é feita entre eles.

O HDFS é um sistema de arquivos feito para facilitar o armazenamento de arquivos muito grandes em clusters feitos com *hardware* de consumo. Isso quer dizer que ele trabalha com arquivos que tenham centenas de *megabytes*, *gigabytes* ou *terabytes*. A utilização de muitos arquivos pequenos é problemática, pois o *NameNode* mantém os metadados do sistema de arquivo na memória, assim bilhões de arquivos pequenos não serão suportados pelo *hardware*. O HDFS foi construído com a ideia de escrever apenas uma vez e ler várias vezes (*write-once, read-many-times*), pensando no fato de que a análise a ser realizada vai envolver grande parte ou todo o conjunto de dados (Shvachko et al., 2010; White, 2012).

2.3.1.1 Arquitetura do HDFS

O HDFS tem dois tipos de nó que operam no modo de mestre-trabalhador. O *NameNode* é o mestre e os muitos *DataNode* são os trabalhadores. O cliente acessa o sistema de arquivos se comunicando tanto com *NameNode* quanto com *DataNode*, mas não precisa conhecê-los, pois se comunica através de uma API (White, 2012). A seguir será exposto o funcionamento das diferentes partes do HDFS.

- *NameNode*

O *NameNode* coordena todo o *namespace*, mantém a árvore que representa o sistema de arquivo, os metadados de todos arquivos e diretórios do sistema. A informação é persistida no disco local em dois arquivos: a imagem do *namespace*, chamada de *checkpoint*, e o *log* de edição dessa imagem, chamado de *journal*. Então ele sabe onde os blocos de um determinado arquivo estão, mas ele não persiste essa informação, ela é recarregada a partir dos *DataNodes*, pois a localização das réplicas pode mudar com o tempo.

- *DataNode*

O *DataNode* mantém e acessa os blocos dos arquivos. Durante a inicialização, cada *DataNode* conecta-se ao *NameNode* e faz um *handshake* com o propósito de verificar o *namespace ID* e a versão do *software*. Se alguma das duas verificações forem diferentes do *NameNode* o *DataNode* é automaticamente desligado. Essas verificações são importantes para manter a integridade do sistema de arquivos.

Depois do *handshake*, o *NameNode* registra o *DataNode* armazenando o seu *storage ID*, que deve ser único. O *storage ID* permite que o *DataNode* seja reconhecido mesmo se reiniciado com um IP diferente ou porta diferente.

- *Heartbeat*

Em operação normal, os *DataNodes* enviam *heartbeats* para o *NameNode*, confirmando que estão operacionais e que os blocos que possuem estão disponíveis. O intervalo padrão de um *heartbeat* é de 3 segundos. Se após

10 minutos o *NameNode* não receber um *heartbeat*, ele considera o *DataNode* como fora de serviço e seus blocos como indisponíveis, iniciando o procedimento de replicação dos blocos em outro *DataNode*.

Os *heartbeats* de um *DataNode* carregam informações referentes a sua capacidade total, fração em uso e o número de transferência de dados em andamento. Possibilitando o *NameNode* cuidar da alocação de espaço e do balanceamento de carga.

O *heartbeat* é o modo como o *NameNode* se comunica com os *DataNode*. Ele responde os *heartbeats* com instruções ao *DataNode*, como replicação de bloco, remoção de bloco, se desligar ou enviar um relatório de blocos. Esses comandos são importantes, pois mantêm a integridade do sistema.

- Bloco

Cada bloco é representado por dois arquivos no sistema de arquivos nativo do *DataNode*, onde o primeiro contém os dados e o segundo mantém os metadados, incluindo *checksum* dos dados e o *generation stamp*. Um *DataNode* envia para o *NameNode* os blocos que possui através de um relatório de blocos. O relatório contém o ID do bloco, o *checksum* e o *generation stamp* de cada réplica de bloco que o servidor hospeda. Esse relatório é enviado imediatamente após o registro do *DataNode* e relatórios subsequentes são enviados de hora em hora.

Um arquivo é dividido em grandes blocos que são armazenados e replicados nos *DataNodes*. Um bloco tem normalmente 128 *megabytes* (o tamanho do bloco é definido pelo usuário) e é replicado em 3 *DataNodes* (pode ser mais, dependendo da configuração do *cluster*). A replicação leva em consideração a topologia do *cluster*, onde o primeiro bloco é alocado em um determinado nó e os próximos blocos são alocados em outros nós de diferentes máquinas. A replicação busca obedecer a seguinte regra: uma máquina não pode ter mais de uma réplica do mesmo bloco.

- Cliente HDFS

Os usuários acessam o sistema de arquivos através do cliente HDFS, onde é

possível ler, criar e deletar arquivos e diretórios. A referência a arquivos e diretórios é feita por meio de caminhos que estão no *namespace*. A aplicação ou o usuário normalmente não sabem que os metadados e armazenamento estão em servidores diferentes, ou que os blocos de cada arquivo têm várias réplicas que se encontram em diferentes servidores.

Quando um cliente lê um arquivo o pedido vai para o *NameNode* que responde com a lista de *DataNodes* que possuem blocos do arquivo. O cliente então contacta o *DataNode* diretamente e faz a requisição da transferência dos blocos necessários. Ao escrever um arquivo, o cliente pede ao *NameNode* uma lista de quais *DataNodes* onde gravar as réplicas do primeiro bloco do arquivo, então um *pipeline* nó a nó é organizado, os dados são enviados e os blocos ao serem gravados enviam uma confirmação ao *NameNode*. Depois que o primeiro bloco foi gravado o cliente faz outra requisição pedindo quais serão os *DataNodes* a serem usados para gravar o próximo bloco, assim um novo *pipeline* é organizado e os dados são enviados. Esse processo se repete até que todo o arquivo seja gravado. Essas interações podem ser vistas na figura 6 (Shvachko et al., 2010).

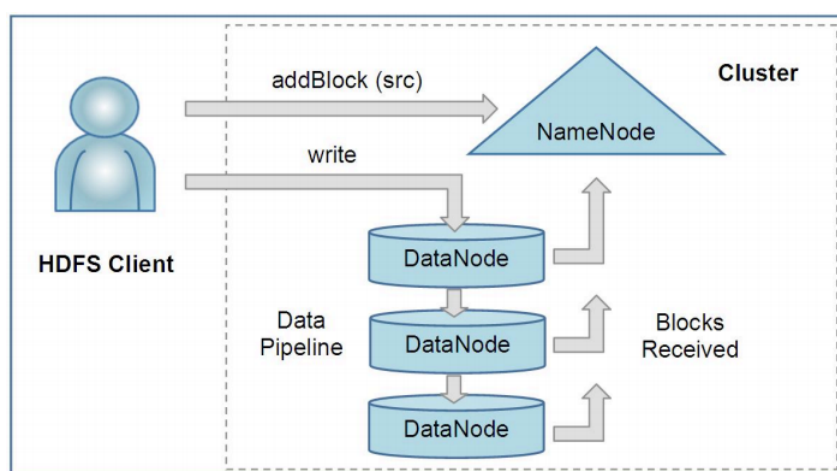


Figura 6: Um cliente HDFS criando um novo arquivo (Shvachko et al., 2010).

Na figura 7 temos o funcionamento em alto nível do HDFS, que mostra o funcionamento e algumas interações já detalhadas. O funcionamento da replicação, os *DataNodes* servindo as requisições de leitura e escrita dos clientes e o

NameNode coordenando as operações de blocos e mantendo os metadados.

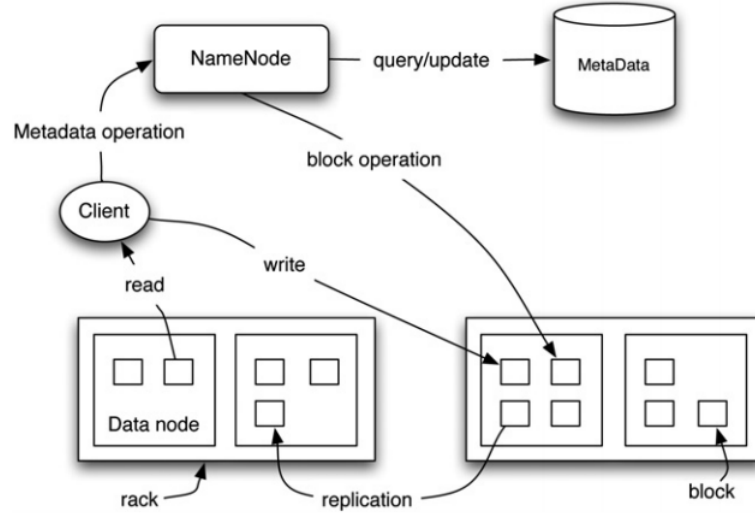


Figura 7: Arquitetura HDFS (Wang et al., 2013).

2.3.2 MapReduce

MapReduce é um modelo de programação para processamento e também geração de grandes conjuntos de dados. O modelo foi inspirado pelas primitivas *map* e *reduce* presentes em linguagens funcionais como *Lisp*. Assim, neste modelo, o desenvolvedor especifica uma função *Map* que processa um par chave/valor e gera um outro conjunto intermediário de pares chave/valor e também especifica uma função *Reduce* que une todos os valores associados com uma determinada chave intermediária (Dean e Ghemawat, 2008; Fang et al., 2014). Conceitualmente:

$$Map(k1, v1) \rightarrow list(k2, v2)$$

$$Reduce(k2, list(v2)) \rightarrow list(v3)$$

Em maiores detalhes a função *Map* é aplicada paralelamente para cada par de chave/valor do domínio inicial, gerando assim uma lista de pares intermediários e iniciando a segunda etapa do processo. A segunda etapa coleta todos os pares com a mesma chave da lista intermediária e cria um grupo de todos os valores para uma determinada chave. Esse novo par de chave/lista de valores é a

entrada que será utilizada na fase de *Reduce*.

A função *Reduce* é então aplicada a cada grupo em paralelo, onde é produzido uma nova coleção com o resultado intermediário por nó de processamento. Esses resultados são então agrupados pelas suas chaves e processados para se chegar em um resultado final.

Para entendimento do paradigma é mostrado o pseudocódigo (algoritmo 1) baseado em *MapReduce* utilizado para o cálculo da maior temperatura anual de uma determinada base de dados. A função *Map* processa cada linha do arquivo de origem emitindo um par ano/temperatura. As temperaturas são agrupadas por ano e são enviados para a função *Reduce* que recebe o ano e sua lista de temperaturas e busca a maior para o determinado ano. A figura mostra o exemplo desse processamento.

Algorithm 1 Funções *MapReduce* usadas para calcular a maior temperatura anual.

```
function MAP(String chave, String valor)
  for all ano e temperatura em valor do
    emitirPar(ano, temperatura)
  end for
end function
function REDUCE(String chaveAno, Lista temperaturas)
  emitirPar(chaveAno, max(temperaturas))
end function
```

O paradigma *MapReduce* pode ser implementado em diferentes arquiteturas, como CUDA, arquitetura com múltiplos processadores, FPGAs, entre outros (Wang et al., 2013).

A implementação do *MapReduce* no *Hadoop* permite que aplicações rodem em um grande cluster construído com *hardware* de baixo custo. Ela tem como enfoque simplificar o processamento de dados paralelos. Ela torna transparente os detalhes de balanceamento de carga, falhas de nós, distribuição e transferência dos dados, permitindo assim que o desenvolvedor se preocupe apenas com o problema a ser resolvido, sem se preocupar com detalhes de paralelização e outros problemas que acompanham o acesso a dados distribuídos (Wang et al., 2013; Dean e Ghemawat, 2010).

Ela também permite que o usuário especifique uma função *combiner*, que é executada na saída de um *map*, resultando na entrada de um *reduce*. Essa função busca minimizar a transferência de dados entre as fases de *map* e *reduce*, diminuindo a quantidade de saídas de dados que entra no *shuffling* (White, 2012). Conceitualmente:

$$Map(k1, v1) \rightarrow list(k2, v2)$$

$$Combiner(k2, list(v2)) \rightarrow list(k2, v2)$$

$$Reduce(k2, list(v2)) \rightarrow list(v3)$$

No exemplo dado no algoritmo 1 a função *combiner* pode ser definida do mesmo modo que a *reduce*, assim o *mapper* local envia apenas a maior temperatura que foi encontrada no *split* para a fase de *shuffling* diminuindo a quantidade de dados a serem processados pelo *reduce* e o tráfego de dados na rede. Essa alteração pode ser vista no algoritmo 2.

Algorithm 2 Funções *MapReduce* usadas para calcular a maior temperatura anual.

```

function MAP(String chave, String valor)
  for all ano e temperatura em valor do
    emitirPar(ano, temperatura)
  end for
end function
function COMBINER(String chaveAno, Lista temperaturas)
  emitirPar(chaveAno, max(temperaturas))
end function
function REDUCE(String chaveAno, Lista temperaturas)
  emitirPar(chaveAno, max(temperaturas))
end function

```

A implementação do *MapReduce* consiste em um único nó mestre que executa o *JobTracker*, que aceita os *Jobs* enviados por um nó cliente, e outros nós *slaves* que possuem uma instância do *TaskTracker*. O *JobTracker* tem a responsabilidade de distribuir o *software* e suas configurações para os nós *slaves*, fazer também o agendamento das tarefas que compõem o *Job* nos *TaskTrackers*, monitorar possíveis falhas e reagendar tarefas nos *TaskTrackers*. Os *TaskTrackers*

executam as tarefas requisitadas pelo *JobTracker*, que é executado em processos Java separados, possibilitando a execução paralela de diversas tarefas em um mesmo nó. A Figura 8 mostra os diferentes componentes do *framework MapReduce* (Wang et al., 2013; Dean e Ghemawat, 2010).

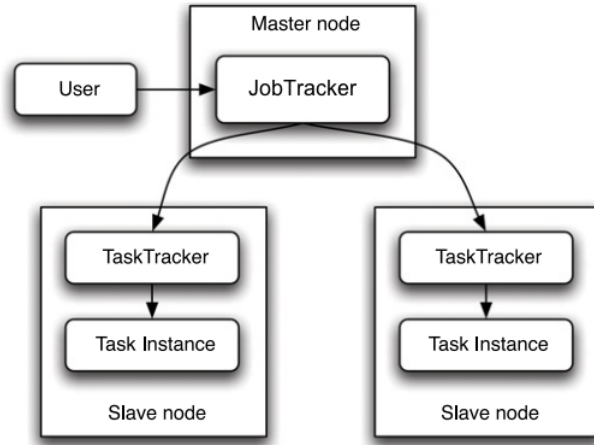


Figura 8: MapReduce no Hadoop. Adaptado de Wang et al. (2013).

A figura 9 mostra o pipeline em alto nível do *Hadoop MapReduce*. Os dados de entrada vêm de arquivos que estão no HDFS. Na fase de mapeamento o arquivo de entrada é dividido em *InputSplits* e cada um é uma unidade de trabalho designado a uma única tarefa *map*. Essa tarefa é associada ao nó que possui os dados referentes ao *InputSplit*, assim cada *map* trabalha com os dados locais, sem depender de tráfego na rede. Quando a fase de mapeamento é concluída, os resultados intermediários são trocados entre todos os nós, sendo agrupados pela chave e se tornando as entradas da fase de redução. O processo de troca dos resultados intermediários é conhecido como *shuffling*. A fase de redução é distribuída nos mesmos nós que a fase de mapeamento e a sua saída é armazenada localmente nos nós *slaves* (Wang et al., 2013).

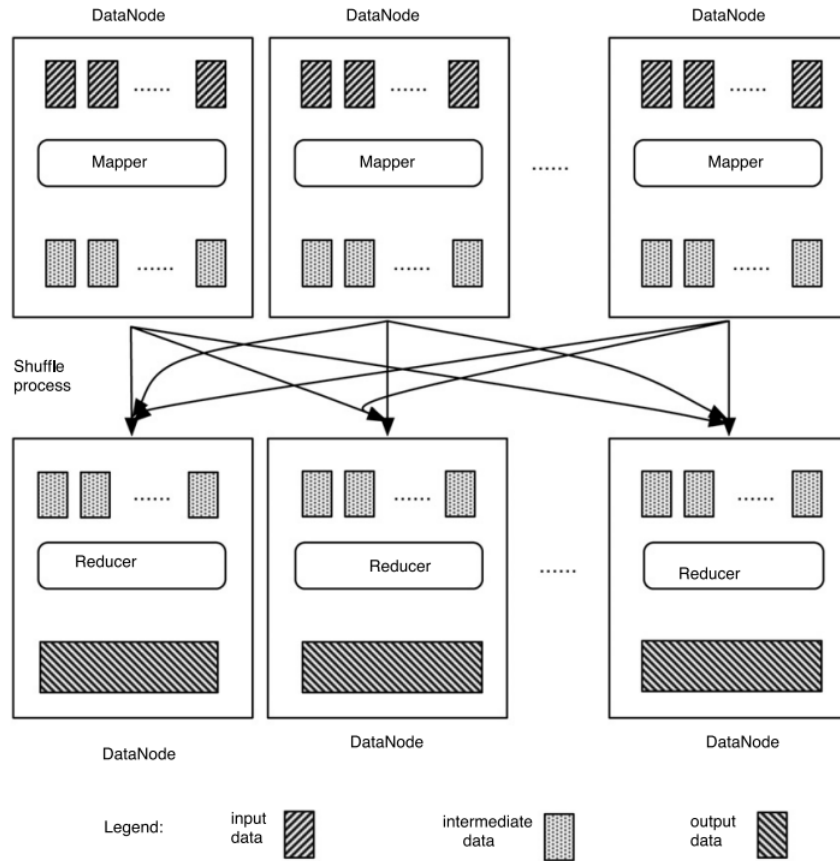


Figura 9: Fluxo de dados no *Hadoop* (Wang et al., 2013).

Neste capítulo foi revisada a bibliografia utilizada na adaptação dos métodos de análise de séries temporais não-lineares utilizando o Hadoop. No próximo capítulo é mostrado como foram adaptados os algoritmos das dimensões fractais e da transformada de *wavelet* para se aproveitar do paralelismo oferecido pelo *MapReduce* no Hadoop.

3 MATERIAIS E MÉTODOS

Este trabalho investiga a viabilidade e o desempenho na adaptação de algoritmos de análise de séries temporais ao modelo *MapReduce*. Para investigar a viabilidade foram selecionadas e implementadas diferentes situações na análise de séries temporais. Os algoritmos implementados foram:

- Transformada de *wavelet*.
- Busca por similaridade usando a distância Euclidiana como função de distância.
- Cálculo da dimensão *Box-Counting*, para atratores com duas dimensões.
- Cálculo da dimensão de correlação, usando reconstrução do espaço de fase.

A próxima seção trata do ambiente que foram desenvolvidos e testados os algoritmos. As próximas seções tratam dos dados utilizados e das peculiaridades de cada implementação respectivamente.

3.1 AMBIENTE DE DESENVOLVIMENTO

O ambiente utilizado foi um *cluster* de 27 máquinas físicas e 5 máquinas virtuais rodando a versão 2.6.0 do software Apache Hadoop. As máquinas virtuais foram utilizadas para a instalação dos serviços principais: o *NameNode*, o *Secondary NameNode* e o *jobHistory*. Nelas também foram instalados serviços que facilitam a administração do *cluster*, como o *Ambari server*, e o cliente de acesso ao HDFS. Elas têm 8GB de memória RAM e 45 GB de HD.

As 27 máquinas físicas foram utilizadas como *DataNodes*, sendo responsáveis por todo o armazenamento de arquivo e processamento. Eles possuem um processador AMD A10 com 4 núcleos e 3,7 GHz de *clock*, 4 GB de memória RAM

e 500 GB de HD. O HDFS está configurado para fazer três replicações de cada bloco, resultando em um total de 4,2 TB de espaço dentro do HDFS.

3.2 DADOS UTILIZADOS

Os algoritmos adaptados foram testados para diferentes conjuntos de dados. Para o algoritmo de busca por similaridade e transformada de *wavelet* foi utilizada uma série de precipitações. Para a dimensão *box-counting* foram utilizados diferentes fractais e figuras geométricas para testar e avaliar os resultados. E para o cálculo da dimensão de correlação foi utilizado um conjunto de pontos reconstruídos a partir da série x do atrator de Lorenz.

3.2.1 Busca por similaridade

Os dados de precipitação utilizados podem ser obtidos no site do projeto TRMM (TRMM, 2015). O arquivo contém séries de precipitação com valores diários para o período de 1998 à 2013 em 2597 pontos diferentes. A transformada de *wavelet* foi utilizada para os dados em 3 granularidades diferentes: mensais, semestrais e anuais. Depois de executada a transformada, foi utilizado a busca por k vizinhos mais próximos nos dados resultantes.

3.2.2 Fractais

Nesta seção serão mostrados os diferentes conjuntos de dados utilizados para testes nos algoritmos de cálculo de dimensão fractal. Na estimativa da dimensão *box-counting* foram utilizados fractais com dimensões previamente conhecidas para comparação dos resultados obtidos. Foram utilizados dois fractais com características diferentes e dois conjuntos geométricos com propriedades conhecidas. No cálculo da dimensão de correlação foi utilizada a série x do atrator de Lorenz (figura 3), utilizando-se a defasagem temporal para gerar as outras dimensões.

3.2.2.1 Dimensão *box-counting*

Foram usados quatro conjuntos diferentes de dados nos testes da dimensão *box-counting*. Os objetos utilizados foram:

- **Reta:** A reta gerada para n pontos é definida pela equação $y = x$. Uma reta possui dimensão fractal analítica igual a um (Jelinek e Fernandez, 1998).
- **Retângulo:** O retângulo gerado possui lados iguais de tamanho n , com um total de n^2 pontos. Um retângulo possui dimensão fractal analítica igual a dois (Jelinek e Fernandez, 1998).
- **Hénon Map:** O mapa de Hénon utilizado é definido pelo sistema de equações 11. Assim como o sistema de Lorenz, ele também é um atrator caótico. O atrator de Hénon tem uma dimensão fractal estimada numericamente de aproximadamente 1,26 para os parâmetros $a = 1,4$ e $b = 0.3$ (Theiler, 1990; Grebogi et al., 1987).

$$\begin{cases} x_{n+1} = 1 - ax_n^2 + y_n \\ y_{n+1} = bx_n \end{cases} \quad (11)$$

A figura 10 mostra o atrator de Hénon gerado para dez mil elementos utilizando os parâmetros $a = 1,4$ e $b = 0.3$ na equação 11.

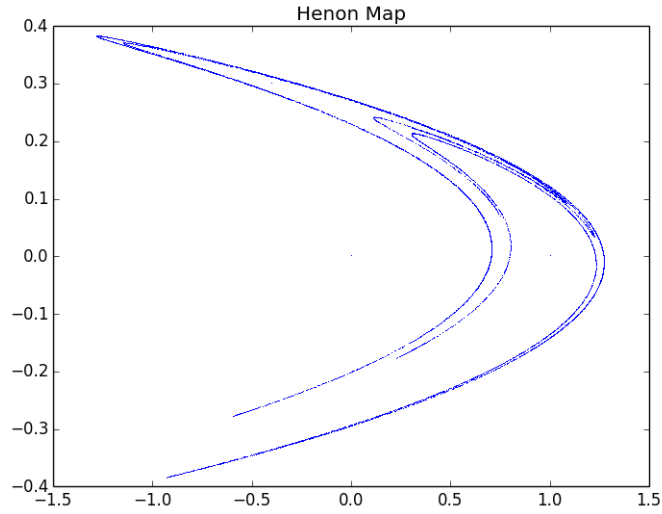


Figura 10: Hénon Map

- **Curva de Koch:** A curva de Koch é um fractal com dimensão $\log(4)/\log(3) = 1,262$. (Bourke, 1993; Jelinek e Fernandez, 1998).

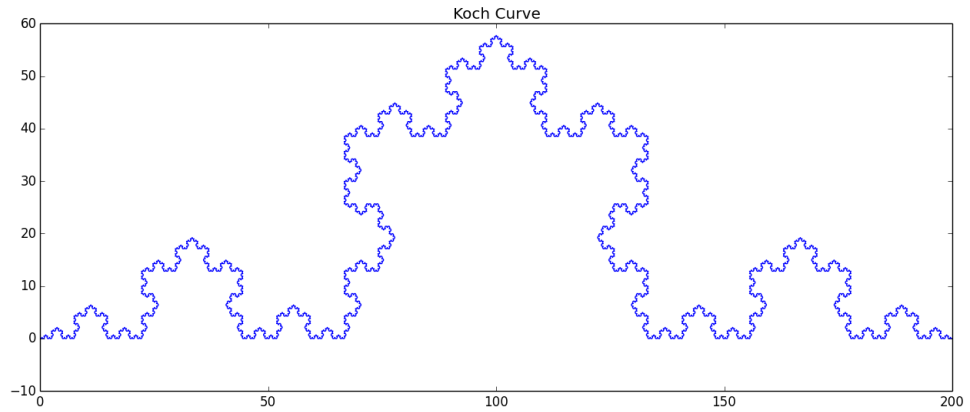


Figura 11: Curva de Koch

3.2.2.2 Dimensão de correlação

O cálculo da dimensão de correlação foi feito sobre o espaço de fase reconstruído a partir de cem mil elementos da série x do atrator de Lorenz (figura 4). Foi utilizado também um arquivo para auxiliar nos parâmetros de paralelização.

3.3 ADAPTAÇÃO DOS ALGORITMOS

Os algoritmos sofreram algumas adaptações para se encaixarem na divisão de fases proporcionada pelo paradigma do *MapReduce*. Essas adaptações serão mostradas nas próximas seções.

3.3.1 *kNN Search*

O algoritmo de busca por vizinhos mais próximos foi adaptado de forma que a série a ser buscada esteja presente em todos os nós da fase de *map*, permitindo que o cálculo da distância entre a série buscada e todas as outras séries do conjunto a ser buscado seja feito de modo paralelo.

Depois de se calcular as distâncias, é necessário ordenar e selecionar as k séries mais próximas baseando-se nas distâncias calculadas, por isso é necessário que os dados sejam comparados entre si, criando uma interdependência entre eles. Esse problema pode ser mitigado ao se buscar os k mais próximos localmente, diminuindo o custo de rede e de um cálculo centralizado, pois se aproveita o paralelismo.

Para efeito de comparação foi implementado uma versão sequencial do algoritmo de *kNN search*. O algoritmo acessa o arquivo que contem a *wavelet* a ser buscada e o carrega para a memória, onde ele será comparado com todas as outras *wavelets* a fim de se determinar as k *wavelets* mais próximas. A medida de distância usada foi a distância euclidiana.

3.3.2 Transformada de *Wavelet*

A transformada de *wavelet* foi aplicada em diversas séries temporais, que tiveram origem em uma única série. O paralelismo oferecido pelo *hadoop* foi utilizado no processo de identificação e divisão da série em outras séries menores. Assim o algoritmo da transformada de *wavelet* não foi paralelizado, o que foi paralelizado é o processo de identificação, preparação dos dados, montagem das séries temporais e parametrização das *wavelets*.

Para demonstrar como a adaptação se comportou, foram definidas três granularidades dos dados na criação das *wavelets*. As diferentes granularidades também mostram diferentes padrões nos dados.

Para comparar foi implementado também uma versão sequencial do processo de identificação e montagem das séries temporais. Foi carregado em um banco de dados uma tabela com os mesmos dados do arquivo usado no *Hadoop*, o algoritmo busca cada uma das séries nessa tabela e aplica a transformada de *wavelet* nos dados de precipitação. A matriz resultado é serializada e grava em um arquivo que é consultado pelo algoritmo que faz a busca de k vizinhos mais próximos.

3.3.3 Dimensão Fractal

Os cálculos de dimensão fractal adaptados possuem diferentes modos de relacionamento entre os dados fazendo com que cada adaptação tenha seus próprios desafios na sua paralelização.

3.3.3.1 Dimensão *Box-Counting*

O algoritmo da dimensão *box-counting* conta quantas caixas de um tamanho ε são necessárias para cobrir um determinado atrator ou série. O paralelismo foi aplicado em cada um dos pontos, onde a adaptação ao ler um determinado ponto define a qual caixa ele pertence, sendo utilizados diferentes tamanhos de caixas.

Depois de determinar as caixas, é necessário que sejam contadas as caixas únicas, pois cada caixa pode ter um ou mais pontos, e todo o processo é executado para os diferentes tamanhos de caixas.

Foi implementado também duas versões sequenciais do algoritmo do *box-counting*, uma que armazena as caixas únicas na memória RAM e outro que armazena em arquivo com auxílio de um banco de dados.

O algoritmo sequencial em memória RAM lê os pontos de um arquivo de entrada, para cada ponto lido define a qual caixa o ponto pertence e o adiciona a um *HashSet*. Para saber quantas caixas únicas forma necessárias para a série

basta saber a quantidade de elementos dentro do *HashSet*, pois o *HashSet* não mantém valores duplicados.

O algoritmo sequencial com auxílio de banco de dados lê os pontos de um arquivo de entrada, para cada ponto lido define a qual caixa o ponto pertence e o adiciona a caixa selecionada a uma tabela. Para saber a quantidade de caixas únicas é feita uma contagem das caixas distintas gravadas na tabela.

3.3.3.2 Dimensão de Correlação

O algoritmo da dimensão de correlação faz uma contagem de quantos pontos estão dentro de um raio r , com origem em todos os pontos da série ou atrator analisado. Essa restrição dificulta a adaptação do algoritmo a um paradigma como o *MapReduce*, pois os dados são dependentes entre si e no *HDFS* os dados estão guardados em diferentes máquinas.

O que foi feito para contornar esse problema foi a cópia da série inteira para cada uma das máquinas envolvidas no processamento e usar um arquivo para controlar e definir o que cada *map* vai processar. Cada linha do arquivo de controle contém a dimensão e o tamanho do raio a ser processado pelo *map* que a leu. Foram executados testes com 1.000, 2.000, 4.000, 8.000, 16.000, 32.000, 64.000 e 100.000 dados.

Neste capítulo foi mostrado como os métodos de análise de séries temporais foram adaptados e implementados no modelo distribuído *MapReduce*. No próximo capítulo os resultados obtidos através da adaptação serão discutidos.

4 RESULTADOS E DISCUSSÃO

As adaptações propostas no capítulo 3 foram implementadas e testadas levando em consideração o tempo que cada tarefa levou para ser executada, a quantidade de registros intermediários que foram gerados e o desempenho total de cada solução.

4.1 IMPLEMENTAÇÃO DA APLICAÇÃO

Os algoritmos adaptados foram implementados em Java utilizando a biblioteca do *Hadoop* para a criação de *jobs MapReduce*. De modo geral, a figura 12 mostra a estrutura da aplicação desenvolvida, que está dividida em pacotes. O pacote *main* contém todas as classes executáveis que iniciam *jobs*.

O pacote *mapper* contém todas as classes que implementam a interface *Mapper* da biblioteca. O pacote *reducer* contém todas as classes que implementam a interface *Reducer* da biblioteca, logo as classes que servem de *combiner* também estão neste pacote. O pacote *writable* contém todas as classes que implementam a interface *Writable* da biblioteca. O pacote *util* contém a classe que realiza a transformada de *wavelet* em uma série temporal.

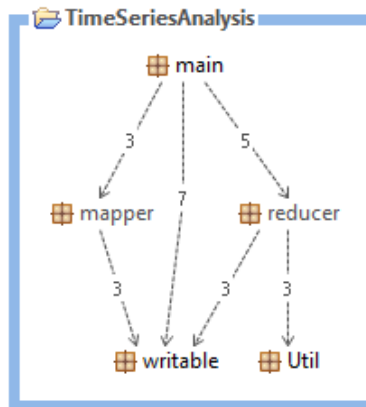


Figura 12: Estrutura da aplicação desenvolvida.

A implementação dos *jobs* desenvolvidos serão mostradas em maiores detalhes nas próximas seções.

4.2 BUSCA POR SIMILARIDADE EM SÉRIES TEMPORAIS

A busca por similaridade depende de duas etapas. Primeiramente, aplica-se a transformada de *wavelet* em todas as séries que serão buscadas, na granularidade necessária. Em seguida, utiliza-se a busca pelos k vizinhos mais próximos (*kNN Search*), utilizando como medida a distância Euclidiana. Em relação aos períodos, foram criadas *wavelets* em três espaços de tempo diferentes: mensal, semestral e anual.

4.2.1 Implementação

Como o problema é dividido em duas etapas, foi desenvolvido para cada etapa um *job MapReduce* específico. O primeiro *job* realiza a transformada de *wavelet* para as séries na base de dados inicial, criando assim um arquivo de saída com todas as *wavelets* criadas. O segundo *job* utiliza como entrada o arquivo de saída do primeiro e uma série temporal escolhida pelo usuário, que é a série a ser buscada pelo *kNN Search*. Nas seções a seguir será explicado em maiores detalhes as implementações.

4.2.2 Tratamento dos Dados

O conjunto de séries temporais utilizado está armazenado em um arquivo CSV, onde cada linha é uma medida de precipitação diária para um determinado local, com formato <Data; Lat; Lon; Prep; ID>. Para facilitar a posterior reconstrução da série, foi utilizado um pré-processamento dos dados para adicionar outros três campos: Month_ID, Semester_ID e Year_ID. Eles foram criados pensando em uma otimização na busca pelas séries originais, então eles são do tipo inteiro para facilitar uma possível busca em um SGBD.

O campo `Month_ID` é formado pela concatenação do ano, mês e local. Ano e mês possuem origem na data de medição e o local é um número que identifica a combinação de latitude e longitude. O campo `Semester_ID` é formado pela concatenação do ano, semestre e local. O campo `Year_ID` é formado pela concatenação do ano e local.

Uma linha agora tem o seguinte formato: `<Data; Lat; Lon; Prep; ID; Month_ID; Semester_ID; Year_ID>`. Esse arquivo servirá de entrada para o *job MapReduce*.

4.2.3 Transformada de *Wavelet*

Aqui é descrito como foi implementada a etapa que aplica a transformada de *wavelet* no conjunto de dados. Ela decompõe o arquivo em séries temporais distintas nas quais será aplicada a transformada de *wavelet*. O processo está ilustrado na figura 13 e foi implementado da seguinte maneira:

- **Map:** Recebe o arquivo de entrada especificado acima e um parâmetro onde o usuário define a granularidade da série a ser estudada. Gerando um registro de saída da seguinte forma: `<ID,Prep>`. Onde ID vai ser um dos três campos definidos e Prep é o valor da precipitação no registro de entrada.
- **Reduce:** Recebe o conjunto de registro de precipitação com uma mesma ID. Neste conjunto ele aplica a transformada de *wavelet*. A saída é um conjunto no formato `<ID,Matriz de Double>`.

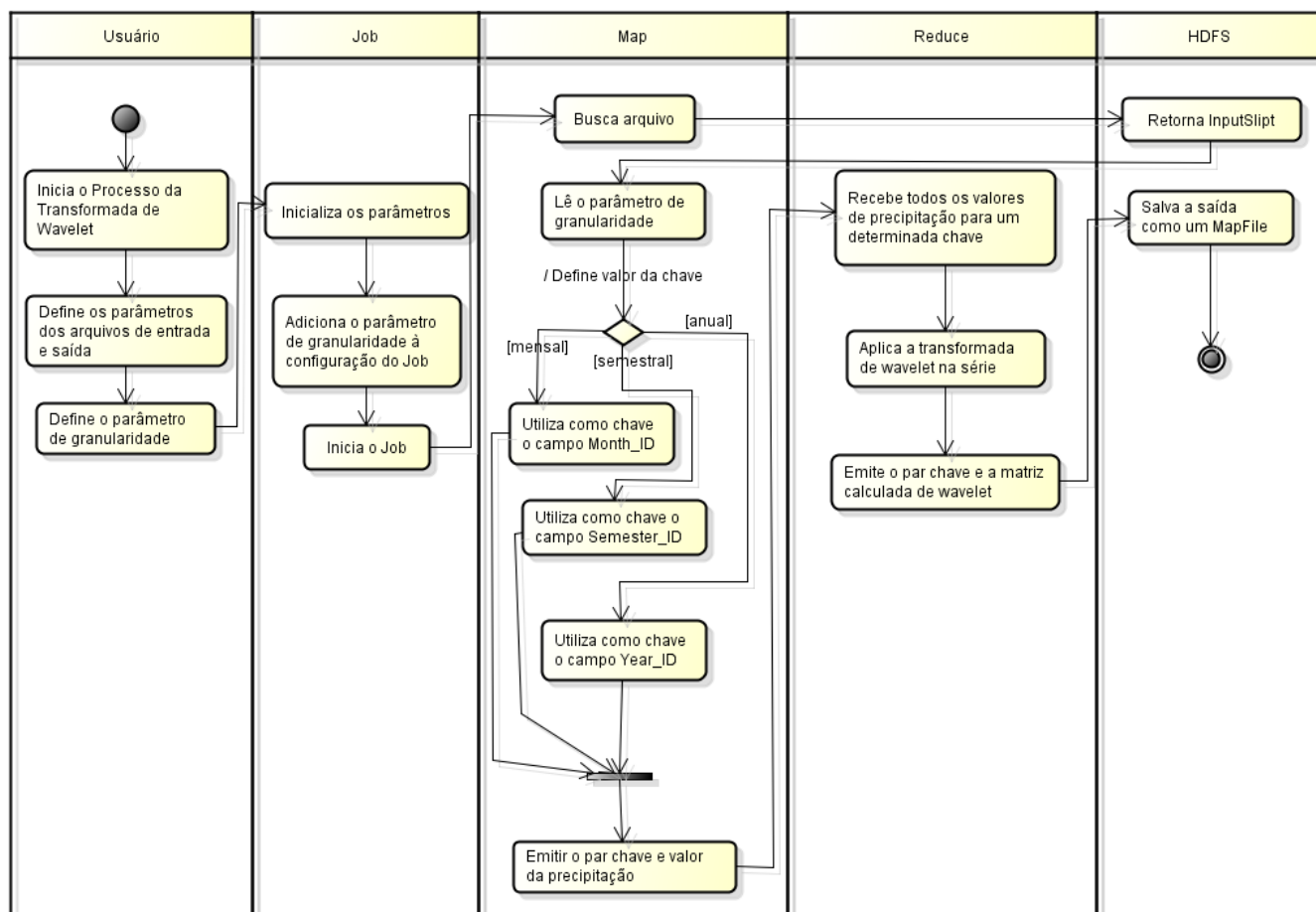


Figura 13: Diagrama de Atividade - Transformada de *Wavelet*

O resultado é gravado em um arquivo no HDFS do tipo *MapFile*. O *MapFile* é um tipo de arquivo ordenado e que permite uma busca por chave, no caso a ID (White, 2012). Esse tipo de arquivo é útil pois permite que o usuário busque uma *wavelet* apenas pela sua ID, o que será importante na próxima etapa.

4.2.4 Criação das *wavelets*

A tabela 4.2.4 mostra a quantidade de registros de precipitação que foram manipulados por cada fase do *job* desenvolvido. Como a fase de *Map* emite um valor para cada registro, a quantidade de registros de entrada é igual a quantidade de registros de saída. Como esta implementação não possui uma fase de *combiner*, a quantidade de registros que servem de entrada para a fase *Reduce* é a mesma da saída do *Map*. Já na saída do *Reduce* existe uma diferença na quantidade de registros para cada uma das granularidades temporais observadas. Esses registros

finais são as *wavelets* criadas pelo *job*.

<i>Wavelet</i>	Entrada no <i>Map</i>	Entrada do <i>Reduce</i>	Saída do <i>Reduce</i>
Mensal	15176868	15176868	496027
Semestral	15176868	15176868	80507
Anual	15176868	15176868	38955

Tabela 1: Registros manipulados na geração das *wavelets*.

A figura 14 compara o tamanho dos arquivos gerados com a quantidade de *wavelets* criadas. Pode-se perceber que a relação quantidade de *wavelets* e tamanho de arquivo foi inversa, significando que as *wavelets* que englobam um período de tempo maior possuem mais dados por *wavelet* do que as que englobam um período menor.

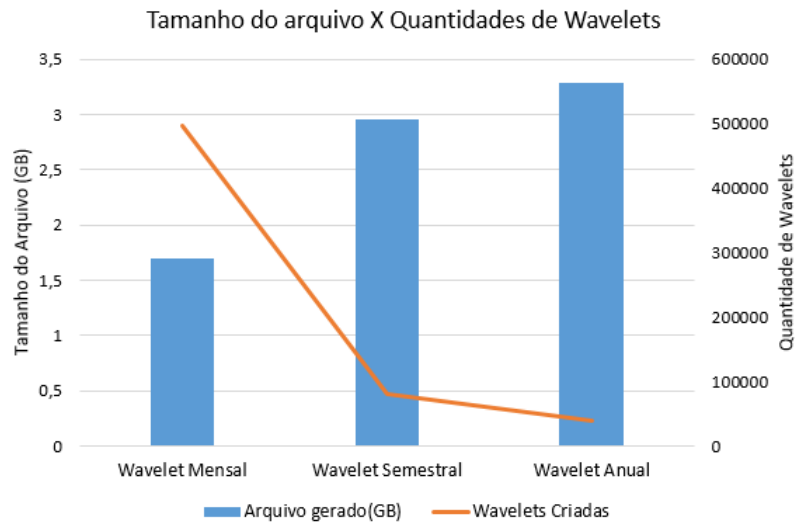


Figura 14: Tamanho dos arquivos criados e a quantidade de registros armazenados.

Na figura 15 é mostrado um comparativo entre os tamanhos médios de cada *wavelet* em cada uma das granularidades propostas. O comparativo mostra que, como visto na figura 14, os tamanhos das *wavelets* aumentam de acordo com o aumento da granularidade temporal.

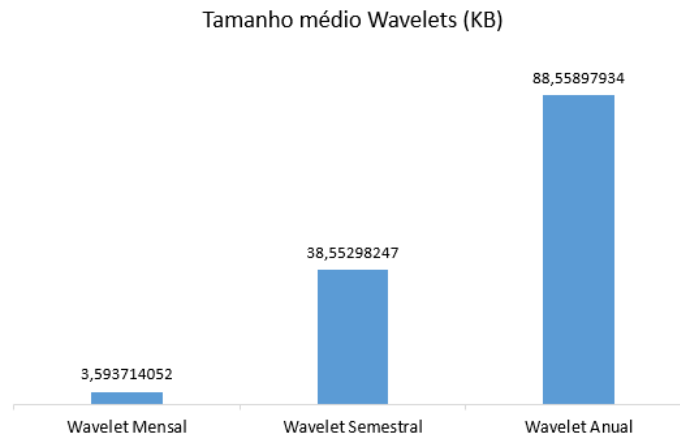


Figura 15: Tamanho médio de cada *wavelet*.

A figura 16 é um comparativo dos tempos de cada fase e tempo total do *job*. É visto que a maior parte do tempo é gasta na fase de *Reduce*. Isso acontece pois é nela onde a série é remontada e passa pela transformada de *wavelet*, gerando os valores a serem gravados.

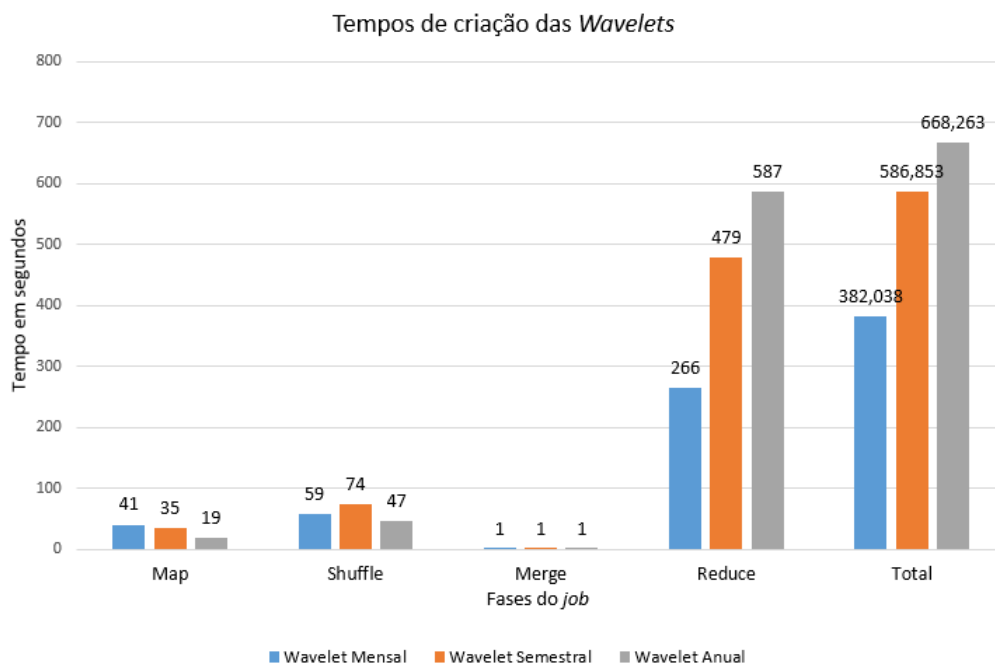


Figura 16: Tempo de cada fase e tempo total comparados.

Na figura 17 pode se ver a diferença de tempo entre a implementação sequencial e a implementação no *Hadoop*.

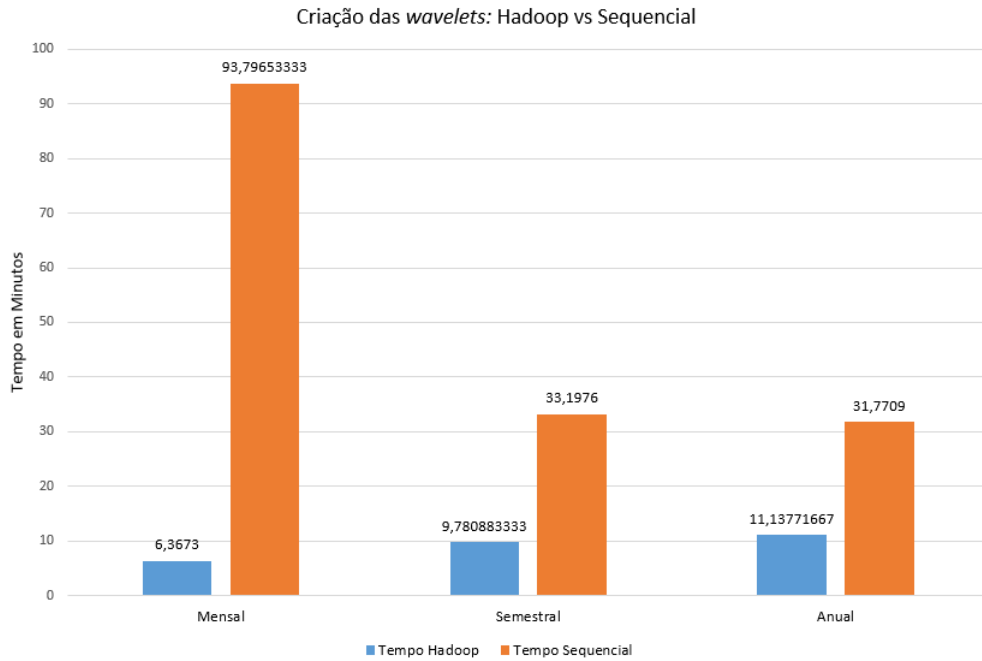


Figura 17: Comparação do tempo sequencial com o tempo do *Hadoop*.

Depois de criadas as *wavelets* é necessário avaliar o desempenho da busca por similaridade usando a técnica k vizinhos mais próximos.

4.2.5 *kNN Search*

Aqui é descrito como foi implementada a etapa que faz a busca por vizinho mais próximos. Essa etapa busca os k vizinhos mais próximos de uma determinada *wavelet*. O usuário define qual série será usada para comparação passando um ID que existe no *MapFile* criado na etapa anterior. Antes de iniciar o processo de *MapReduce* o *job* busca o ID escolhido e o copia para um arquivo temporário que é adicionado ao *cache* distribuído para ser copiado a todas as máquinas do *cluster*. O processo *MapReduce* está ilustrado na figura 18 e foi implementado da seguinte forma:

- **Map:** Tem como entrada <ID, Matriz de Double> proveniente do *MapFile* da etapa anterior e o arquivo temporário. Faz o cálculo da distância Euclidiana entre as duas matrizes e emite como chave o ID que foi selecionado pelo usuário e está no arquivo temporário e como valor o ID proveniente

do *MapFile* e a distância calculada. Gera assim um registro de saída no formato <ID_temp, ID Distância>.

- **Combiner:** Recebe o conjunto de registro com as distâncias já calculadas e emite apenas as k mais próximas da série a ser buscada. Tem a saída no mesmo formato que o *map*.
- **Reduce:** Faz a mesma coisa que o *combiner*, mas muda o formato do registro de saída. Emitindo apenas o ID e os IDs das *wavelets* mais próximas.

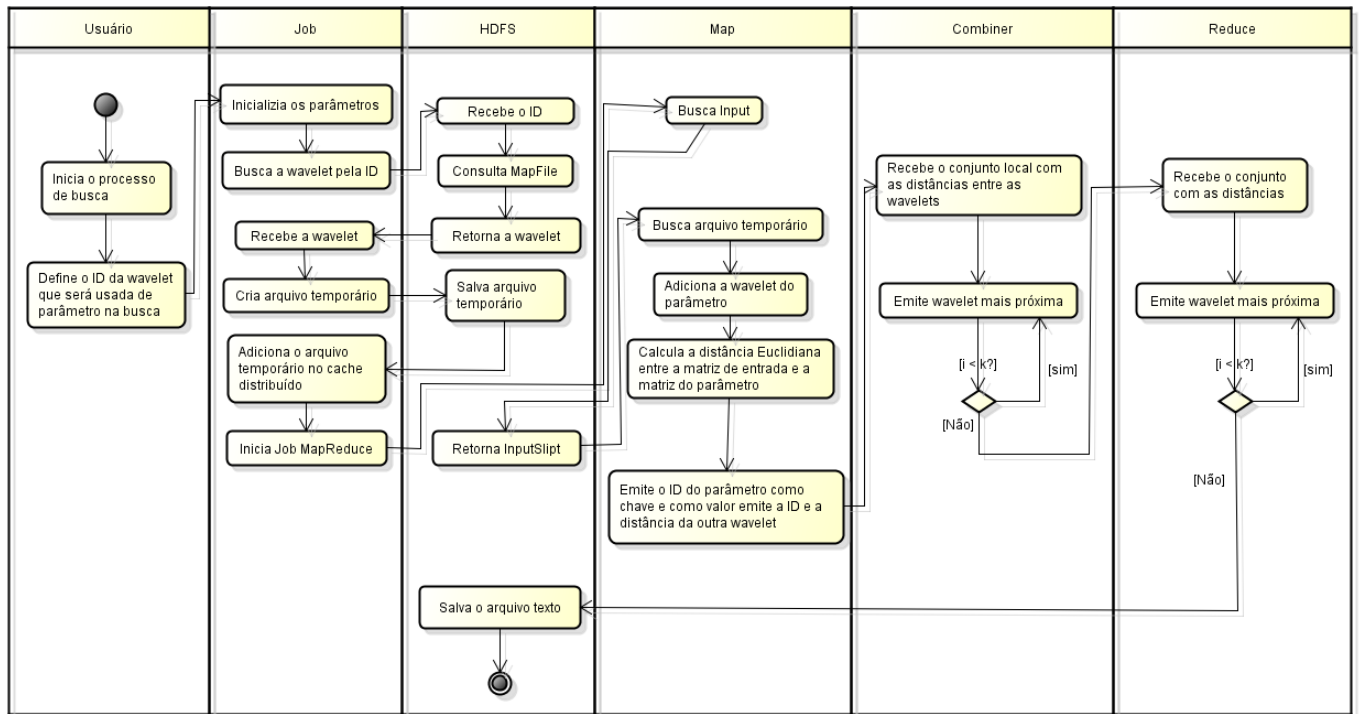


Figura 18: Diagrama de Atividade - *kNN Search*

Nesse caso, a função *combiner* diminui a quantidade de tráfego na rede, pois de cada *Mapper* são emitidos apenas k registros, diminuindo o impacto na fase de *shuffling*. O resultado é gravado em um arquivo texto no HDFS e contém as k séries mais próximas da selecionada.

4.2.6 Busca em *wavelets*

A tabela 4.2.6 mostra quantidade de registros manipulados por cada fase da busca por similaridade nas diferentes granularidades de tempo. As quantida-

des de entradas da fase *Map* são as *wavelets* resultantes do *job* de criação. A quantidade de registros de entrada da fase *Reduce* é a quantidade que saiu da fase *Combiner*. Estes valores estão diretamente ligados à quantidade de tarefas de mapeamento iniciadas pelo *Hadoop*, pois a fase *Combiner* funciona como uma redução local, diminuindo o tráfego de dados na rede. O *Reduce* produz um único registro com as k IDs mais próximas separadas por espaços.

<i>kNN Search</i>	Entrada no Map	Entrada do Combiner	Entrada do Reduce	Saída do Reduce
Mensal	496027	496027	140	10
Semestral	80507	80507	240	10
Anual	38955	38955	270	10

Tabela 2: Registros manipulados no *kNN search*.

A figura 19 é um comparativo dos tempos de cada fase da busca por similaridade nas *wavelets*. Nela é visto que a maior parte do tempo é gasta na fase de *shuffle*. Essa fase é de agrupamento pelas mesmas chaves, é uma fase de organização.

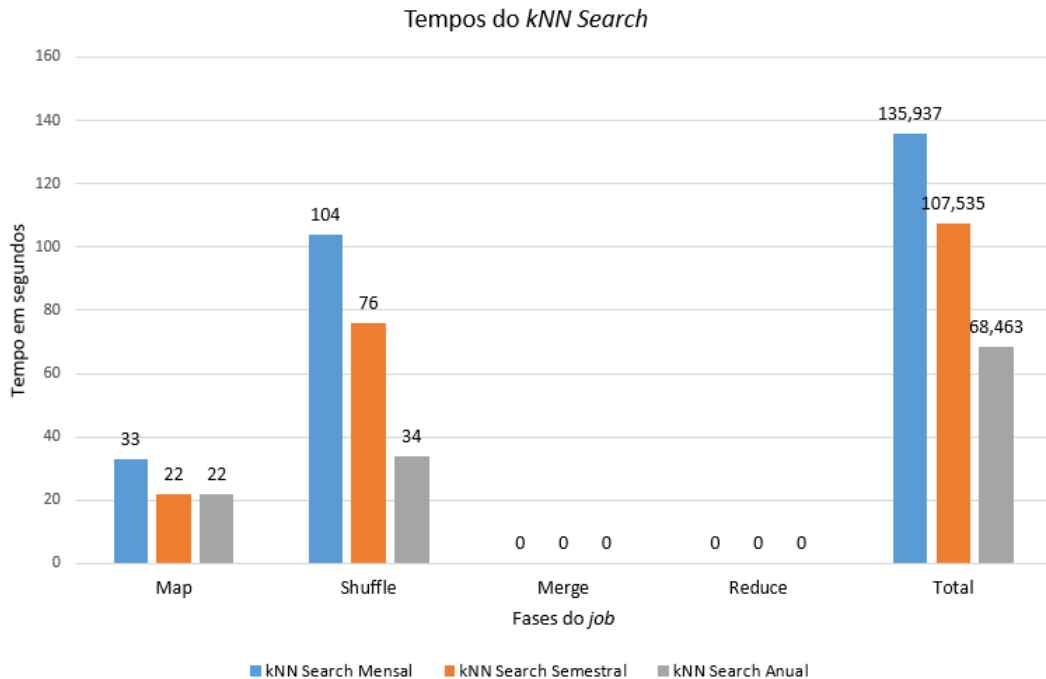


Figura 19: Tempo de cada fase e tempo total comparados.

A figura 20 mostra os tempos da implementação sequencial e a implementação no *Hadoop*.

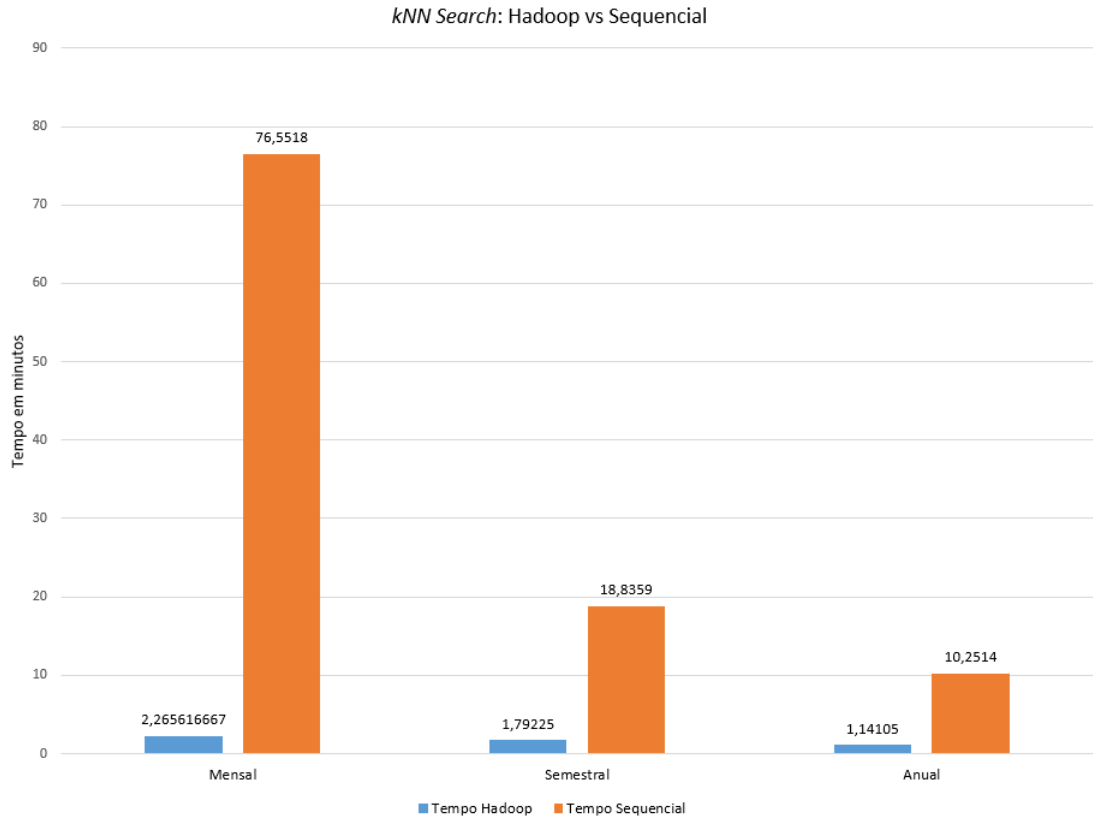


Figura 20: Comparação do tempo do sequencial com o tempo do *Hadoop*.

4.3 SÉRIES TEMPORAIS CAÓTICAS

No estudo das séries temporais caóticas foi implementado o cálculo de duas dimensões fractais diferentes, a dimensão *box-counting* e a dimensão de correlação. Elas utilizam diferentes formatos nos conjuntos de dados iniciais, a primeira necessita de um arquivo inicial com uma lista de pontos em duas dimensões e a segunda necessita de um arquivo contendo a distância entre todos os pontos.

4.3.1 Dimensão Box-Counting

O *job* da dimensão *box-counting* espera que cada linha do arquivo de entrada contenha um par de valores que define um ponto em um espaço bidimensional. O algoritmo conta a quantidade de caixas de um tamanho ε necessárias para cobrir a série estudada e faz isso para caixas de diferentes tamanhos.

4.3.1.1 Implementação

O cálculo foi implementado em um *job MapReduce*. Onde o *map* emite para cada ponto as caixas que ele pertence e o *reduce* faz a contagem da quantidade de caixas que foram utilizadas para cobrir a série. A contagem é feita por tamanho de caixa, então na saída do *job* tem um registro com a contagem para ε utilizado. O processo está ilustrado na figura 21 e ficou da seguinte forma:

- **Map:** Tem como entrada um ponto $\langle x, y \rangle$. Para cada tamanho de ε é emitida uma caixa que contém o ponto. Então, para um único ponto, são emitidos n registros, no formato $\langle \varepsilon_n, \text{Box_ID} \rangle$.
- **Combiner:** Recebe o conjunto com as caixas e emite os registros únicos, no mesmo formato que o *map*.
- **Reduce:** Recebe o conjunto com as caixas. Conta a quantidade de caixas únicas necessárias para cobrir a série, para cada um dos tamanhos, emitindo pares no formato $\langle \log(1/\varepsilon), \log N(\varepsilon) \rangle$.

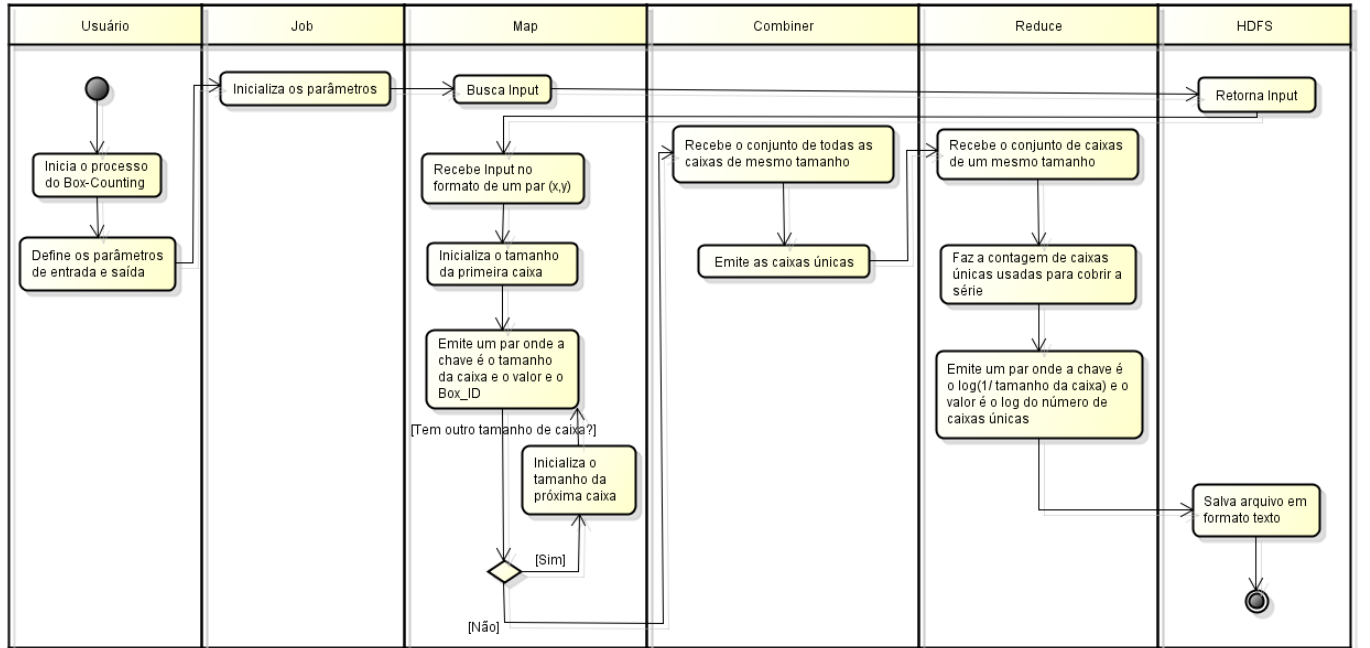


Figura 21: Diagrama de Atividade - *Box-Counting*

A função *combiner* busca diminuir a quantidade de registros duplicados enviados para o *shuffling* que serão descartados no *reducer*, pois uma caixa pode conter mais de um ponto e apenas a caixa é contada.

Essa primeira implementação proposta (figura 21) foi testada para três conjuntos de dados com 1 milhão, 10 milhões e 100 milhões de pontos, respectivamente. Para 1 milhão de pontos, o tempo total de cálculo foi de 2 minutos e 6 segundos. Para os outros dois conjuntos de dados, a implementação não se comportou da maneira esperada, pois a quantidade de caixas criadas para cobrir a série de teste foi muito alta. Assim as fases de *combiner* e *reduce* tiveram problemas de falta de memória na tentativa de identificar e contar as caixas únicas.

Para resolver o problema de escalabilidade foi proposta e desenvolvida uma nova implementação que divide o trabalho em dois *jobs MapReduce*:

- **Job 1:** Gera um conjunto com todas as caixas únicas dos tamanhos definidos pelo usuário.
- **Job 2:** Faz a contagem de caixas únicas por tamanho e emite um par no formato $\langle \log(1/\varepsilon), \log N(\varepsilon) \rangle$ para cada tamanho ε de caixa.

A figura 22 mostra um comparativo do tamanho do arquivo gerado pelo primeiro *job* para diferentes quantidades de pontos. Esses arquivos gravam as caixas únicas para cada diferente tamanho de caixa. Pelos tamanhos pode-se perceber o problema da implementação anterior, ela tentava manter em memória todas essas informações em um *hashmap*. Ao se dividir o problema em duas partes, a implementação de cada *job* foi simplificada, saindo de um *job* para dois *jobs* mais simples.

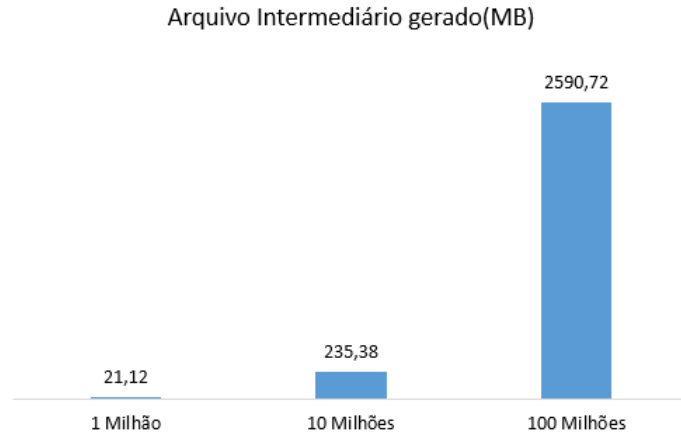
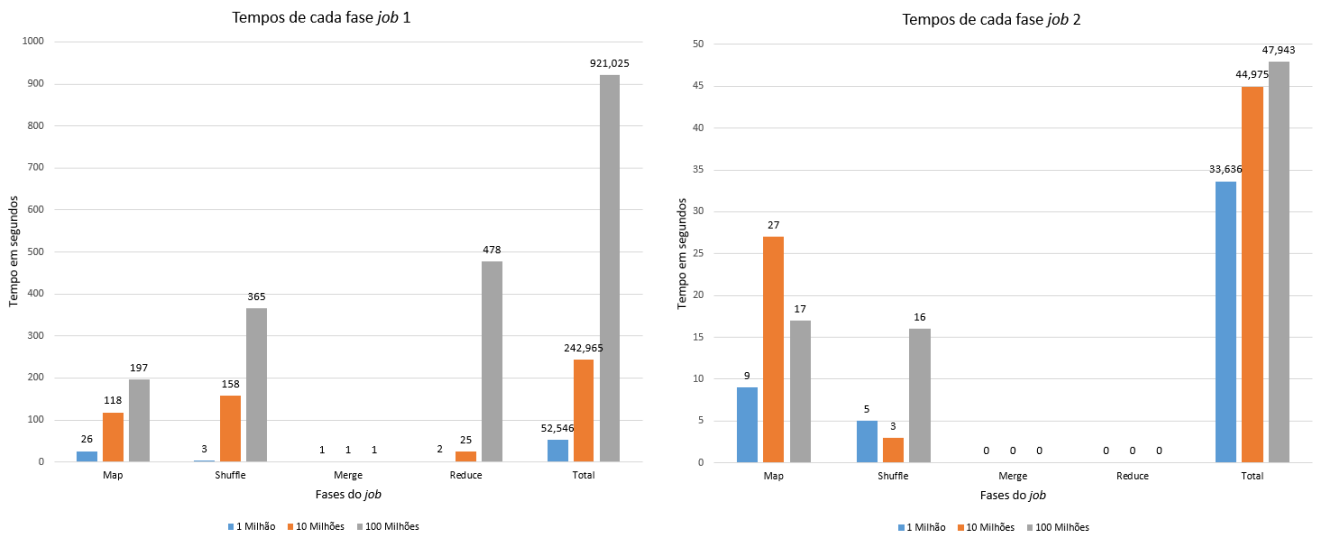


Figura 22: Tamanho dos arquivos intermediários gerados pelo primeiro *job* para diferentes quantidades de pontos.

O arquivo final varia entre 300 - 400 *Bytes*. O arquivo intermediário guarda todas as caixas únicas por tamanho de caixa. O arquivo final guarda os pontos da curva formada por $\log N \times \log(1/\varepsilon)$.

A figura 23 é um comparativo dos tempos de cada fase dos *jobs* envolvidos no cálculo da dimensão *box-counting*. Nela percebe-se que o maior gasto de tempo está na primeira parte, que é a parte do *job* que gera o conjunto com todas as caixas únicas por tamanho.



(a) Tempo de cada fase para o primeiro *job*.

(b) Tempo de cada fase para o segundo *job*.

Figura 23: Tempos de cada *job* comparados.

Na próxima seção será demonstrado como o algoritmo implementado se comportou com diferentes quantidades de dados.

4.3.1.2 Teste de escalabilidade - Retas

O conjunto de dados utilizados nestes testes foi a reta. Ele é gerado conforme o capítulo 3. Foram utilizados dez tamanhos de caixas diferentes, com tamanhos variando de um até noventa e um, incrementados de dez em dez. Foram utilizados dez conjuntos de dados diferentes, iniciando em dez milhões de pontos e terminando em cem milhões.

A figura 24 mostra o comportamento do tempo total, a união dos dois *jobs*, em relação a quantidade de dados e o tamanho dos arquivos que armazenam esses dados.

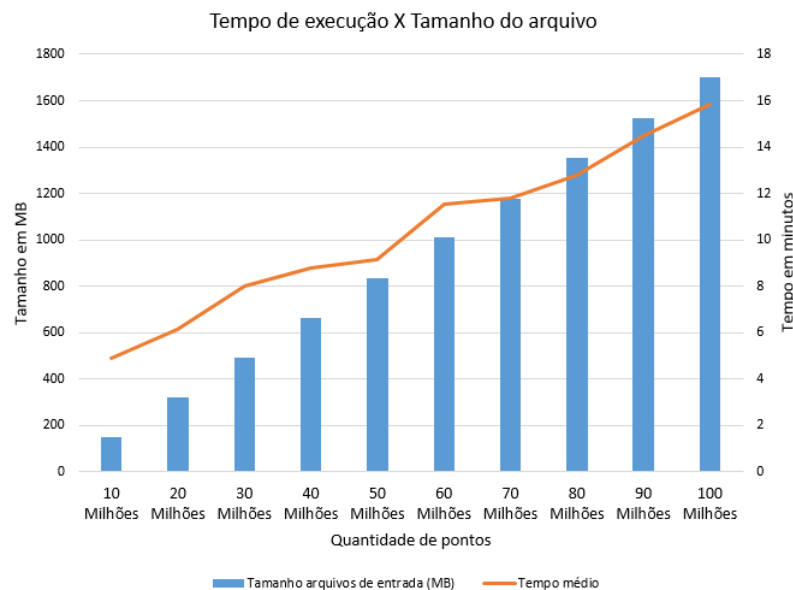
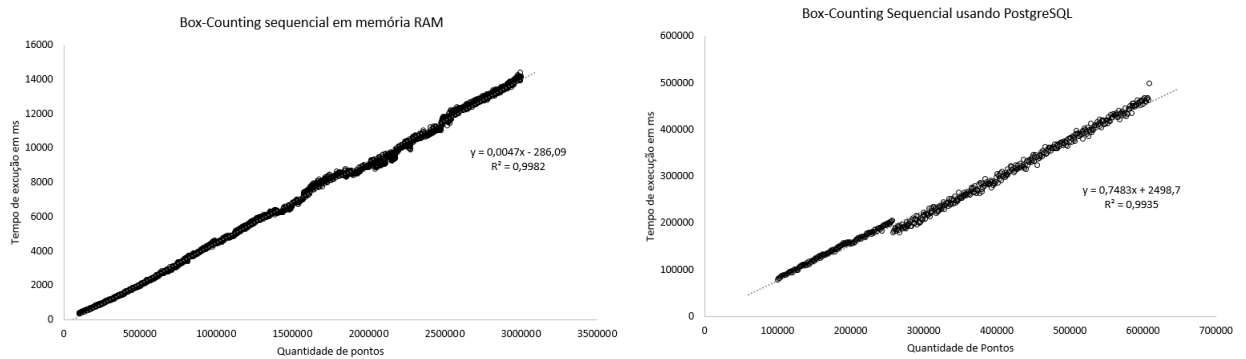


Figura 24: Tempos de execução para arquivos diferentes.

Para comparar a implementação do *hadoop* com a implementação sequencial do *box-counting*, os algoritmos sequenciais foram executados em uma máquina nó do *cluster*. As implementações sequenciais sofreram diferentes problemas, as duas não foram escaláveis, a implementação em memória não conseguiu chegar a uma quantidade de dados igual ao *hadoop* e a implementação que utiliza-se do *PostgreSQL* demora muito tempo para executar. Para contornar esses problemas

foi gerado um conjunto de execuções para cada um dos algoritmos. Com essas execuções foram criadas as regressões da Figura 25. As regressões foram usadas para estimar o comportamento do algoritmo sequencial para as quantidades de dados usadas no *hadoop*.



(a) Regressão linear para o algoritmo em memória RAM. (b) Regressão linear para o algoritmo usando o *PostgreSQL*.

Figura 25: Regressões criadas para as execuções sequenciais.

A Figura 26 compara o tempo de execução da implementação no *hadoop* com o tempo de execução estimado para cada uma das implementações sequenciais.

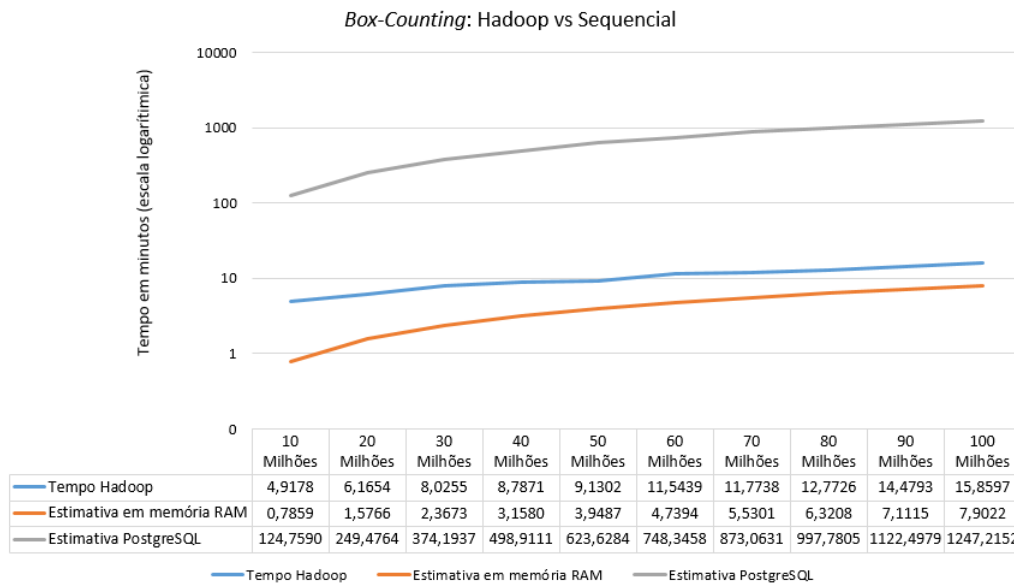


Figura 26: Tempos de execução para o *hadoop* e os sequenciais estimados.

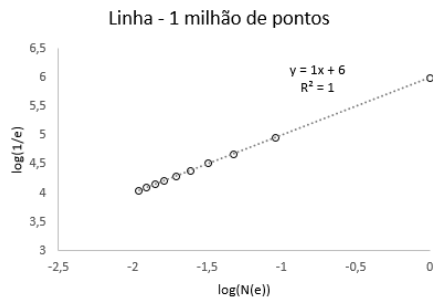
A próxima seção mostra os resultados obtidos pela implementação do

hadoop para diferentes conjuntos de dados e diferentes quantidades de dados.

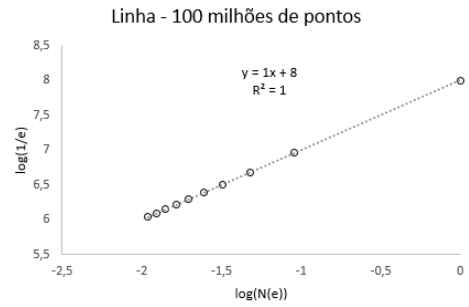
4.3.1.3 Dimensões calculadas

No cálculo da dimensão foram usados conjuntos nos quais suas dimensões já são conhecidas para que o resultado obtido pela implementação possa ser comparado a resultados reais.

As figuras 27 mostram o valor estimado pela implementação para uma reta. Nos dois casos o valor calculado foi o mesmo e não houve variação do valor esperado. Nos dois casos foram usados os mesmos tamanhos de caixa.



(a) Dimensão *Box-Counting* calculada com 1 milhão de pontos

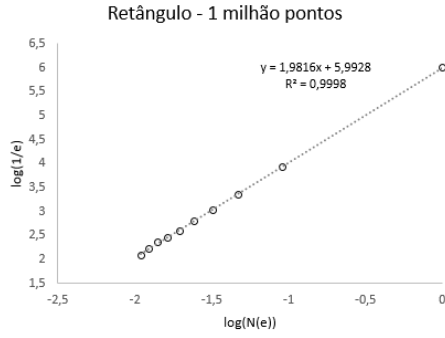


(b) Dimensão *Box-Counting* calculada com 100 milhões de pontos

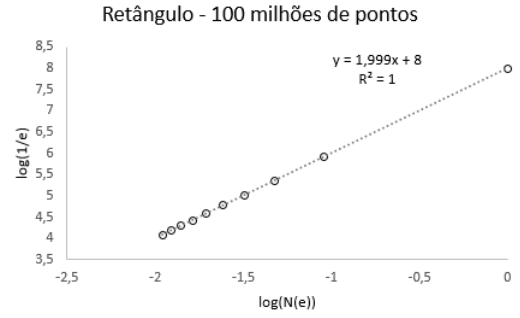
Figura 27: Dimensão *Box-Counting* calculada para uma reta.

As figuras 28 mostram os valores estimados pela implementação para um retângulo. Aqui os valores estimados foram diferentes para diferentes quantidades de pontos no conjunto de dados iniciais.

Para um milhão de pontos foi estimado um valor de 1,9816 para a dimensão *box-counting*. Para cem milhões de pontos foi estimado um valor de 1,999. Nos dois casos o desvio foi menor que 1% em relação a dimensão dois esperada de um retângulo.



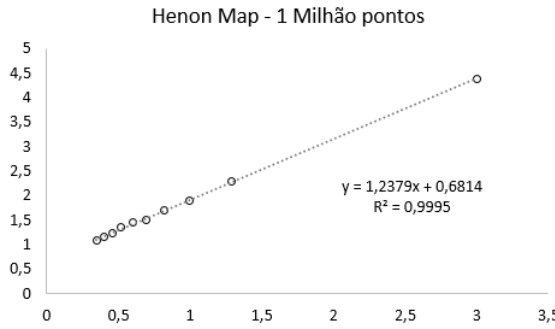
(a) Dimensão *Box-Counting* calculada com 1 milhão de pontos



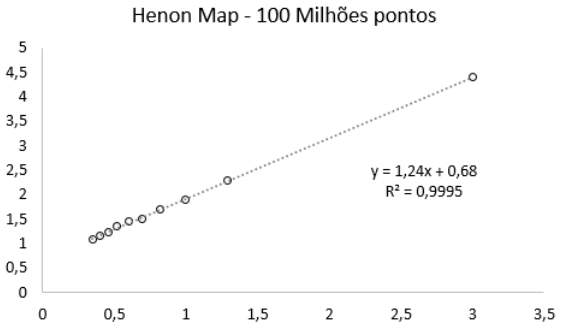
(b) Dimensão *Box-Counting* calculada com 100 milhões de pontos

Figura 28: Dimensão *Box-Counting* calculada para um retângulo.

Os gráficos da figuras 29 mostram as curvas $\log N(\varepsilon) \times \log(1/\varepsilon)$ e a dimensão estimada para o mapa de Hénon. Com uma maior quantidade de pontos a estimativa chegou mais perto da dimensão esperada de 1,26, embora a diferença tenha sido muito pequena entre as duas.



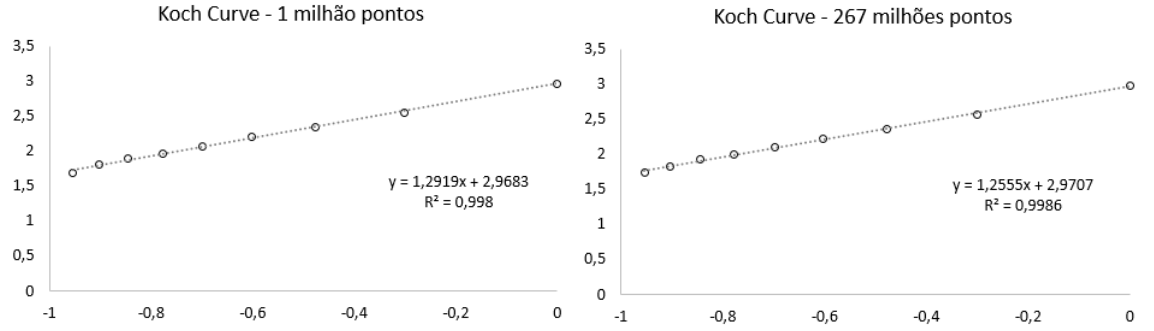
(a) Dimensão *Box-Counting* calculada com 1 milhão de pontos



(b) Dimensão *Box-Counting* calculada com 100 milhões de pontos

Figura 29: Dimensão *Box-Counting* calculada para o mapa de Hénon.

Os gráficos da figura 30 mostram os valores estimados para a dimensão *box-counting* para a curva de Koch. A estimativa do conjunto de dados com 1 milhão de pontos tem um erro de 2% em relação ao valor esperado de 1,262, já a estimativa com 267 milhões de pontos tem um erro menor de 1%.



(a) Dimensão *Box-Counting* calculada com 1 milhão de pontos (b) Dimensão *Box-Counting* calculada com 267 milhões de pontos

Figura 30: Dimensão *Box-Counting* calculada para a curva de Koch.

4.3.2 Dimensão de Correlação

O cálculo da dimensão de correlação depende da interação entre todos os pontos do atrator a ser analisado, como pode ser visto na equação 8. A abordagem utilizada foi a criação de um arquivo de controle com a dimensão e o tamanho do raio a ser utilizado no processo de *MapReduce*.

4.3.2.1 Tratamento dos Dados

Foram utilizados dois arquivos no cálculo da dimensão. O primeiro arquivo é a série temporal a ser analisada e o segundo é um arquivo com os parâmetros utilizados pelo *map* para controlar qual parte do cálculo será feita.

Como série temporal foi utilizada nos 100.000 primeiros elementos do x da equação de Lorenz. Esse arquivo foi adicionado ao HDFS e copiado para todas as máquinas envolvidas no processamento através do *cache* distribuído. Cada linha do arquivo possui o valor de x .

O arquivo de parâmetro foi construído com as dimensões e os diferentes tamanhos de raios usados, cada linha do arquivo contém um valor de dimensão e tamanho de raio, separados por um espaço em branco. Então para 9 dimensões e 40 tamanhos de raios diferentes é criado um arquivo com 360 linhas. Esse arquivo que é o *input* do processamento *MapReduce*.

4.3.2.2 Implementação

Como dito anteriormente o arquivo de parâmetros é criado e utilizado como o *input* do *job MapReduce* e o arquivo com a série temporal é adicionado ao *cache* distribuído. Com o auxílio do parâmetro o *map* sabe qual até qual dimensão a série vai ser reconstruída e utiliza o tamanho do raio para calcular a equação 8.

No *Hadoop* a quantidade de *mappers* é definida pelo *job* de acordo com o tamanho do arquivo de entrada. No caso dessa implementação foi utilizado outro método de particionar o arquivo de entrada o *NLineInputFormat*. Esse método de particionamento o arquivo de acordo com um número N , onde as N linhas serão processadas em uma mesma máquina. No caso do arquivo de entrada com 360 linhas, com $N = 5$, cria 72 *mappers*, se $N = 1$ seriam 360 *mappers*. O processo ficou da seguinte forma:

- **Map:** Tem como entrada o registro com o valor da dimensão e tamanho do raio. Faz o cálculo da correlação (Equação 8) para o tamanho do raio especificado. Retorna como chave o tamanho do raio e como valor a dimensão e o valor da correlação calculado.
- **Reduce:** Recebe o conjunto de resultados para cada raio e tem como saída o tamanho do raio e as correlações para as dimensões reconstruídas.

O diagrama de atividade na figura 31 mostra com mais detalhes os fluxos do processo especificado. Mostra que fica a cargo do usuário a criação do arquivo de controle e definição dos parâmetros de execução.

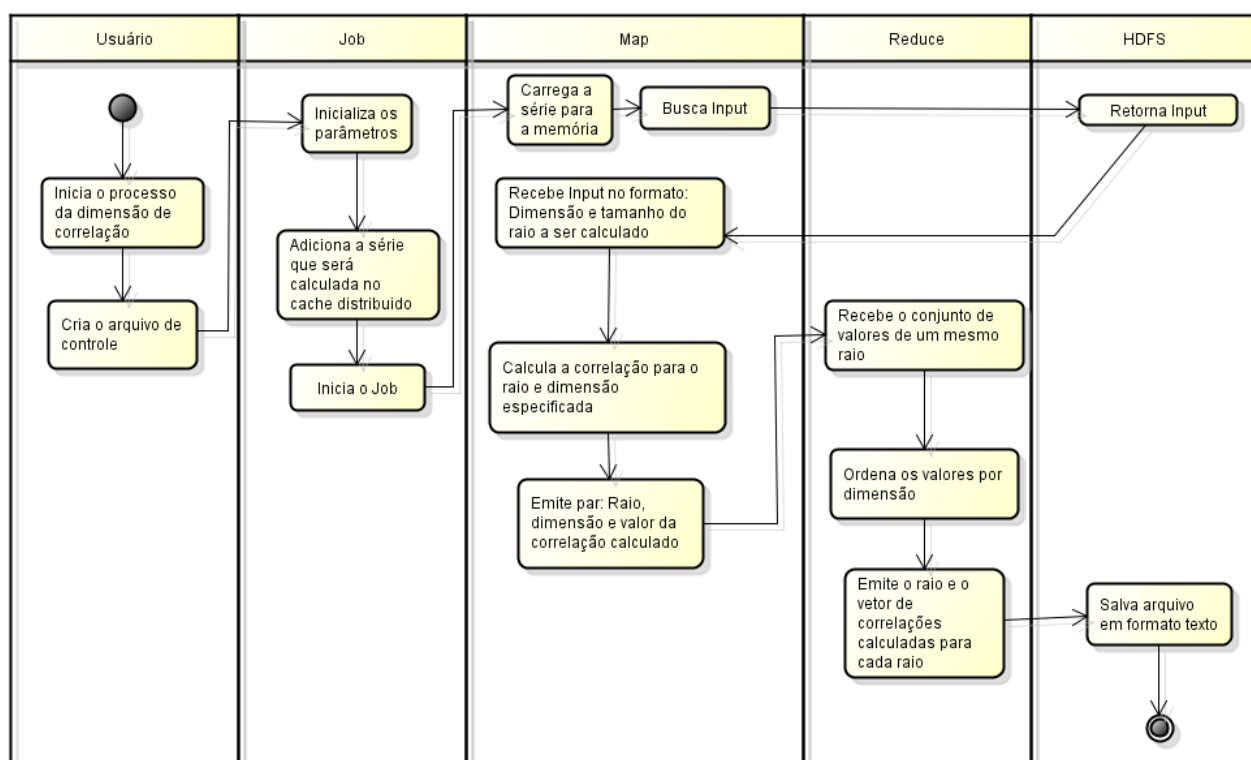


Figura 31: Diagrama de Atividade - Dimensão de Correlação

O problema com a abordagem é a necessidade de que a série precisa estar em todas as máquinas, para séries muito grandes isso se torna um gargalo e demanda mais recursos como memória e processamento.

Para testar a implementação foram criados dois arquivos de controle diferentes e feita uma execução sequencial do algoritmo. Um arquivo com as especificações já citadas onde é definido em cada linha uma dimensão e um tamanho de raio e outro com apenas as dimensões em cada uma das linhas.

O arquivo com apenas as dimensões calcula o valor da correlação (Equação 8) para cada um dos raios definidos, nos teste foram utilizadas 9 dimensões, que resultou na criação de 9 *maps*. O resultado dele esta nomeado nas figuras 32 e 33 por "Hadoop divisão por dimensão".

O arquivo com as dimensões e tamanhos de raios definidos esta nominado pela quantidade de *maps* que foram inicializados. Esse arquivo foi utilizado com diferentes divisões na quantidade de linhas processados por cada *mapper*. Ele foi dividido em 5, 10 e 40 linhas por *mapper*, essas divisões geraram execuções com

72,36 e 9 *mappers* respectivamente. Os resultados obtidos estão nas figuras 32 e 33.

A figura 32 mostra os tempos de execuções das implementações no *Hadoop* e da implementação sequencial. Os algoritmos foram executados para 1.000, 2.000, 4.000, 8.000, 16.000, 32.000, 64.000 e 100.000 dados.

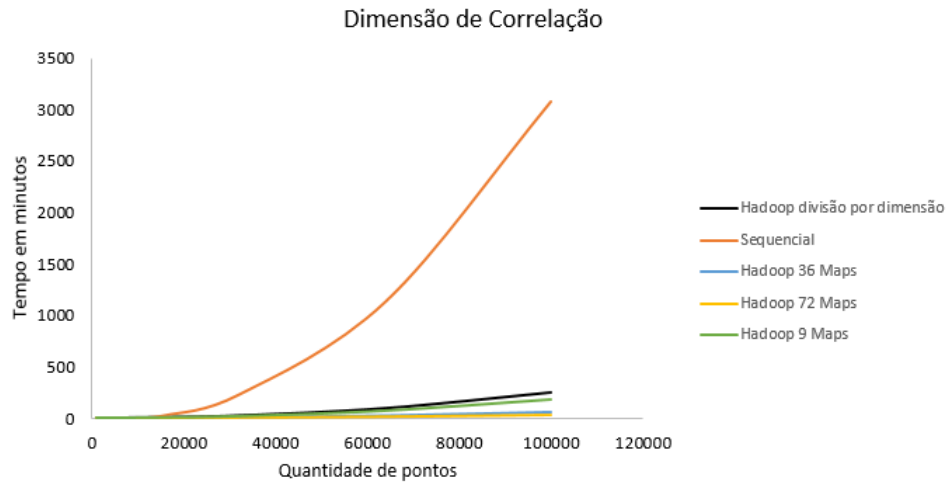


Figura 32: Tempo de execução para diferentes quantidades de pontos.

A figura 33 mostra o detalhe da figura 32, ela mostra os tempos de execuções das implementações do *Hadoop* para 1.000, 2.000, 4.000, 8.000 e 16.000 pontos e mostra os tempos da execução sequencial para 1.000, 2.000, 4.000 e 8.000 pontos. Com esse detalhamento é perceptível que até 2.000 pontos a execução sequencial é mais eficiente que o *Hadoop*, mas que depois de 2.000 pontos as implementações no *Hadoop* são mais eficientes. Da para notar também que a partir de 8.000 pontos quanto maior a quantidade de *maps* maior a eficiência da implementação.

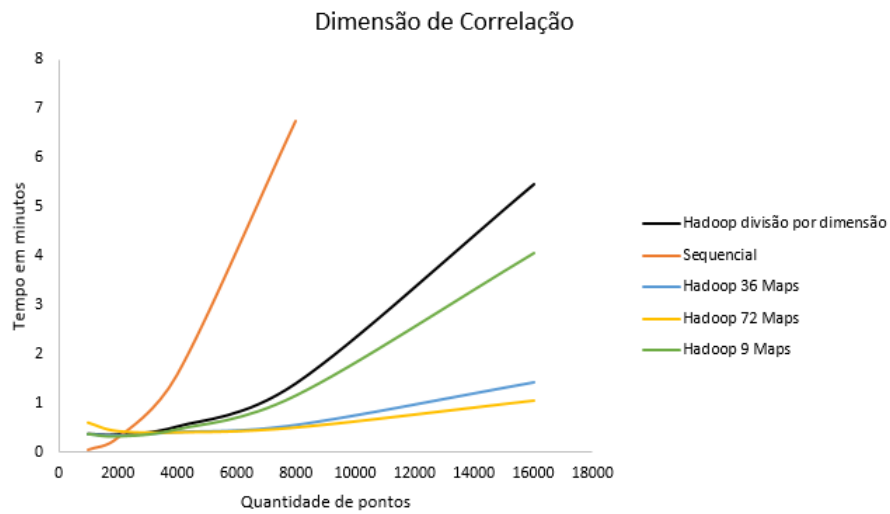


Figura 33: *Zoom* na parte inicial da Figura 32.

Neste capítulo foi demonstrada a adaptação das análises de séries temporais propostas. Também é mostrado as facilidades e dificuldades de se adaptar os problemas a um novo paradigma. O próximo capítulo trata das conclusões deste trabalho e de possíveis trabalhos futuros.

5 CONSIDERAÇÕES FINAIS

Neste trabalho foram adaptados diferentes formas de análises de séries temporais em uma arquitetura distribuída. Iniciando pela aplicação da transformada de *wavelet* em diversas séries. Os dados criados pela transformada foram utilizados para se fazer uma busca por similaridade no domínio de frequência para isso, foi utilizado uma busca por k vizinhos mais próximos tendo como função de distância a distância euclidiana.

A estimativa da dimensão *box-counting* mostrou que o paradigma permite uma maior flexibilidade na resolução de problemas ao se utilizar vários *jobs MapReduce* encadeados. Essa flexibilidade resultou em um algoritmo mais simples de ser implementado e que foi robusto o suficiente para não falhar onde a primeira implementação falhou.

O cálculo da dimensão de correlação mostrou uma fraqueza do *MapReduce*. Como o paradigma facilita o acesso aos dados isolados dentro de um *map*, isso se tornou um problema, pois impossibilita a comparação com dados em outros *maps*. O problema enfrentado foi como resolver a interdependência dos dados para que eles possam processados dentro de um ou mais *jobs MapReduce*. A solução foi pré-calcular a distância entre todos os pontos do conjunto de dados, o que resultou na geração de um arquivo de distâncias que possui crescimento exponencial de seu tamanho a medida que é aumentada a quantidade de pontos analisados.

Ao comparar as implementações da dimensão *box-counting* e a dimensão de correlação fica evidente que o problema de interdependência dos dados existe no cálculo da dimensão de correlação e não afeta muito o cálculo da dimensão *box-counting*.

Assim fica demonstrado como o contexto do *Big Data* pode se associar e resolver diversos problemas na área de pesquisa ambiental, como o armazenamento e processamento de diversos dados não estruturados com origens em

diferentes sensores.

O *MapReduce* fornece um modo de processar e analisar esses dados. Ele demanda adaptações nos algoritmos a um diferente paradigma com o potencial de simplificar os algoritmos se utilizando de diferentes tarefas *MapReduce* encadeadas.

5.1 CONTRIBUIÇÕES

Este trabalho contribui ao utilizar uma ferramenta de *Big Data* no contexto de pesquisa ambiental. O software Apache Hadoop facilitou o armazenamento de grandes quantidades de dados não estruturados de forma distribuída e facilitou também o processamento desses dados através do *MapReduce*.

Outra contribuição foi a demonstração da flexibilidade que o *MapReduce* proporciona no processamento de dados ambientais. O encadeamento de *jobs* pode resolver problemas complexos de modo simplificado. Também fica como contribuição que, mesmo sendo um paradigma flexível, ele não é ideal para as situações onde os dados são muito interdependentes.

Uma contribuição mais direta foi a adaptação dos algoritmos da transformada de *wavelet*, da busca por k vizinhos mais próximos e os algoritmos da dimensão fractal, facilitando a sua aplicação em grandes quantidades de dados.

5.2 TRABALHOS FUTUROS

Os estudos realizados neste trabalho permitem melhorias em suas adaptações de algoritmos para arquiteturas distribuídas, através do aprofundamento do estudo dessas arquiteturas. Os estudos podem incluir novas soluções para o problema de interdependência dos dados em adaptações de algoritmos.

Pode-se melhorar a fase de busca por similaridade implementando outras medidas de distâncias entre as séries, como o *dynamic time warping* (DTW). A busca pode ser otimizada com a utilização de índices locais e a utilização de *z-ordering*, como foi feito no artigo de Zhang et al. (2012). O artigo trata

de um *kNN join* mas pode se buscar uma adaptação para o *kNN Search* ou em um *akNN Search*, que criaria a matriz de distância entre todos as *wavelets* calculadas. É possível também implementar outras *wavelets* mães para se utilizar na transformada.

O trabalho também abre possibilidade de implementação e adaptação de outras análises que envolvam dados ambientais, que não foram mencionadas ou discutidas neste trabalho.

REFERÊNCIAS

- Baker, G. L. (1996). *Chaotic dynamics: an introduction*. Cambridge University Press.
- Bolzan, M. J. A. (2006). Transformada em ondaleta: Uma necessidade. 28(4):563–567.
- Bourke, P. (1993). Fractal dimension calculator user manual. *Swinburne University of Technology Web*.
- Chan, K.-P. e Fu, A.-C. (1999). Efficient time series matching by wavelets. In *Data Engineering, 1999. Proceedings., 15th International Conference on*, páginas 126–133. IEEE.
- Cuzzocrea, A., Song, I.-Y., e Davis, K. C. (2011). Analytics over large-scale multidimensional data: the big data revolution! In *Proceedings of the ACM 14th international workshop on Data Warehousing and OLAP*, páginas 101–104. ACM.
- Darilmaz, I. (2006). *Wavelet based similarity measurement algorithm for seafloor morphology*. Thesis, Massachusetts Institute of Technology.
- Dean, J. e Ghemawat, S. (2008). MapReduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113.
- Dean, J. e Ghemawat, S. (2010). MapReduce: a flexible data processing tool. *Communications of the ACM*, 53(1):72–77.
- Fang, W., Sheng, V. S., Wen, X., e Pan, W. (2014). Meteorological data analysis using MapReduce. *The Scientific World Journal*, 2014.
- Ghemawat, S., Gobioff, H., e Leung, S.-T. (2003). The google file system. In *ACM SIGOPS Operating Systems Review*, volume 37, páginas 29–43. ACM.

- Gomes, R. S. R. (2012). *Arquitetura de processamento paralelo com unidade de processamento gráfico de propósito geral para aplicação em séries temporais de dados ambientais*. Tese de Doutorado, PPGFA.
- Grassberger, P. e Procaccia, I. (1983). Characterization of strange attractors. *Physical review letters*, 50(5):346.
- Grebogi, C., Ott, E., e Yorke, J. A. (1987). Chaos, strange attractors, and fractal basin boundaries in nonlinear dynamics. *Science*, 238(4827):632–638.
- Jelinek, H. F. e Fernandez, E. (1998). Neurons and fractals: how reliable and useful are calculations of fractal dimensions? *Journal of neuroscience methods*, 81(1):9–18.
- Lorenz, E. N. (1963). Deterministic nonperiodic flow. *Journal of the atmospheric sciences*, 20(2):130–141.
- Machiwal, D. e Jha, M. K. (2012). *Hydrologic time series analysis: Theory and practice*. Springer Science & Business Media.
- McAfee, A. e Brynjolfsson, E. (2012). Big data: the management revolution. *Harvard business review*, páginas 60–6.
- Morettin, P. A. e Toloi, C. M. d. C. (1981). *Modelos para previsão de séries temporais*, volume 1. Instituto de matematica pura e aplicada.
- Nunes, S. A., Romani, L. A., Avila, A. M., Traina Jr, C., de Sousa, E. P., e Traina, A. J. (2010). Análise baseada em fractais para identificação de mudanças de tendências em multiplas séries climáticas.
- Ong, H. Y. (2010). *Accelerating Data-Intensive Applications Using MapReduce*. Tese de Doutorado, University of Edinburgh.
- Santos, C. A. G., Freire, P. K. d. M. M., e Torrence, C. (2013). A Transformada de Wavelet e sua Aplicação na Análise de Séries Hidrológicas. 18:271–280.

- Shvachko, K., Kuang, H., Radia, S., e Chansler, R. (2010). The hadoop distributed file system. In *Mass Storage Systems and Technologies (MSST), 2010 IEEE 26th Symposium on*, páginas 1–10. IEEE.
- Takens, F. (1981). Detecting strange attractors in turbulence. In *Dynamical systems and turbulence, Warwick 1980*, páginas 366–381. Springer.
- Theiler, J. (1990). Estimating fractal dimension. *JOSA A*, 7(6):1055–1073.
- Torrence, C. e Compo, G. P. (1998). A practical guide to wavelet analysis. *Bulletin of the American Meteorological society*, 79:61–78.
- Torrence, C. e Webster, P. J. (1999). Interdecadal changes in the ENSO-monsoon system. *Journal of Climate*, 12:2679–2690.
- TRMM (2015). Mission overview. [Online; Acesso em: 21/03/2015].
- Vieira, M. R., FIGUEIREDO, J. M. d., Liberatti, G., e Viebrantz, A. F. M. (2012). Bancos de dados NoSQL: conceitos, ferramentas, linguagens e estudos de casos no contexto de big data. *SIMPÓSIO BRASILEIRO DE BANCO DE DADOS. São Paulo*.
- Wang, L., Tao, J., Ranjan, R., Marten, H., Streit, A., Chen, J., e Chen, D. (2013). G-Hadoop: MapReduce across distributed data centers for data-intensive computing. *Future Generation Computer Systems*, 29(3):739–750.
- White, T. (2012). *Hadoop: The definitive guide*. "O'Reilly Media, Inc."
- Zaslavsky, A., Perera, C., e Georgakopoulos, D. (2013). Sensing as a service and big data. *arXiv preprint arXiv:1301.0159*.
- Zhang, C., Li, F., e Jestes, J. (2012). Efficient Parallel kNN Joins for Large Data in MapReduce. In *Proceedings of the 15th International Conference on Extending Database Technology, EDBT '12*, páginas 38–49, New York, NY, USA. ACM.

Apêndice A: Implementação Transformada de Wavelet

A.1 Mapper

```
package mapper;

import org.apache.hadoop.io.DoubleWritable;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapred.JobConf;
import org.apache.hadoop.mapred.Mapper;
import org.apache.hadoop.mapred.OutputCollector;
import org.apache.hadoop.mapred.Reporter;

import java.io.IOException;

public class WaveletTransformMapper implements Mapper<
    LongWritable, Text, LongWritable, DoubleWritable> {

    static int periodo;

    @Override
    public void map(LongWritable longWritable, Text value,
        OutputCollector<LongWritable, DoubleWritable>
```

```

outputCollector, Reporter reporter) throws
IOException {
    String line = value.toString();
    line = line.trim();
    String s[] = line.split(";");
    long key = Long.parseLong(s[periodo]);

    //YearMonthLocation yearMonthLocation = new
        YearMonthLocation();
    if(s.length !=8){
        //erro
    }
    double prep = Double.parseDouble(s[3]);
    outputCollector.collect(new LongWritable(key), new
        DoubleWritable(prepare));
}

@Override
public void close() throws IOException {

}

@Override
public void configure(JobConf jobConf) {
    String tipo = jobConf.get("tipo");
    switch (tipo){
        case "mensal":
            periodo = 5;
            break;
        case "semestral":
            periodo = 6;

```

```

        break;
    case "anual":
        periodo = 7;
        break;
    default:
        periodo = 5;
        break;
    }
}
}

```

A.2 Reducer

```

package reducer;

import Util.Wavelet;
import org.apache.hadoop.io.DoubleWritable;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.mapred.JobConf;
import org.apache.hadoop.mapred.OutputCollector;
import org.apache.hadoop.mapred.Reducer;
import org.apache.hadoop.mapred.Reporter;
import writable.DoubleTwoDArrayWritable;

import java.io.IOException;
import java.util.ArrayList;
import java.util.Iterator;
import java.util.List;

public class WaveletTransformReducer implements Reducer<
    LongWritable, DoubleWritable, LongWritable,

```

```

DoubleTwoDArrayWritable> {

    static int periodo;

    @Override
    public void reduce(LongWritable key, Iterator<
        DoubleWritable> values, OutputCollector<
        LongWritable, DoubleTwoDArrayWritable>
        outputCollector, Reporter reporter) throws
        IOException {
        DoubleWritable valueIn;

        List<Double> waveletInput = new ArrayList<Double
            >();
        double sum = 0;
        while (values.hasNext()){
            valueIn = values.next();
            double val = valueIn.get();
            waveletInput.add(val);
            sum += val;
        }
        if (waveletInput.size() == periodo){ // Cria as
            matrizes com o mesmo numero de colunas

            Wavelet wavelet = new Wavelet(waveletInput);
            double[][] result = wavelet.waveletTransform()
                ;
            if(result.length > 0){
                DoubleWritable[][] resultWritable = new
                    DoubleWritable[result.length][result
                        [0].length];
            }
        }
    }
}

```

```

        for (int i = 0; i < result.length; i++) {
            for (int j = 0; j < result[0].length;
                j++) {
                if (sum > 0){ // Verfica se a
                               transformada nao esta
                               retornando NaN
                    resultWritable[i][j] = new
                        DoubleWritable(result[i][j]
                            );
                }else{ // Se sum == 0, a matriz e
                               zerada.
                    resultWritable[i][j] = new
                        DoubleWritable(0);
                }
            }
        }
        DoubleTwoDArrayWritable out = new
            DoubleTwoDArrayWritable(resultWritable)
            ;
        outputCollector.collect(key,out);
    }
}

@Override
public void close() throws IOException {

}

@Override
public void configure(JobConf jobConf) {

```

```
String tipo = jobConf.get("tipo");
switch (tipo){
    case "mensal":
        periodo = 28;
        break;
    case "semestral":
        periodo = 182;
        break;
    case "anual":
        periodo = 365;
        break;
    default:
        periodo = 28;
        break;
}
}
}
```

Apêndice B: Implementação kNN Search

B.1 Mapper

```
package mapper;

import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.FileSystem;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.*;
import org.apache.hadoop.mapreduce.Mapper;
import org.apache.hadoop.util.ReflectionUtils;
import writable.DoubleTwoDArrayWritable;

import java.io.IOException;
import java.net.URI;

public class KNNSearchMapper extends Mapper<LongWritable,
    DoubleTwoDArrayWritable, LongWritable, Text> {

    static String outputKey;
    static DoubleWritable[][] wavelet_searched = null;
    public static final Log LOG = LogFactory.getLog(
        KNNSearchMapper.class);
```

```

@Override
protected void setup(Context context) throws
IOException, InterruptedException {
    if (wavelet_searched == null) {
        URI[] localPaths = context.getCacheFiles();
        Configuration conf = context.getConfiguration
            ();
        try {

            FileSystem fs = FileSystem.get(conf);
            SequenceFile.Reader reader = new
                SequenceFile.Reader(fs, new Path(
                    localPaths[0]), conf);

            Writable key = (Writable) ReflectionUtils.
                newInstance(reader.getKeyClass(), conf)
                ;
            DoubleTwoDArrayWritable value = (
                DoubleTwoDArrayWritable)
                ReflectionUtils.newInstance(reader.
                    getValueClass(), conf);

            while (reader.next(key, value)) {
                outputKey = key.toString();
                wavelet_searched = (DoubleWritable
                    [][]) value.toArray();
            }

        } catch (IOException e) {
            throw new RuntimeException(e);
        }
    }
}

```

```

        }
    }
}

@Override
protected void map(LongWritable key,
    DoubleTwoDArrayWritable value, Context context)
    throws IOException, InterruptedException {
    DoubleWritable[][] values = (DoubleWritable[][])(
        value.toArray());
    double sum = 0;
    for (int i = 0; i < values.length; i++) {
        for (int j = 0; j < wavelet_searched[0].length
            ; j++) {
            sum += Math.pow((values[i][j].get() -
                wavelet_searched[i][j].get()), 2);
        }
    }
    context.write(new LongWritable(Long.parseLong(
        outputKey)), new Text(key + "\t" + Math.sqrt(
        sum)));
}

}

```

B.2 Combiner

```

package reducer;

import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.Text;

```

```

import org.apache.hadoop.mapreduce.Reducer;

import java.io.IOException;
import java.util.TreeMap;

public class KNNSearchCombiner extends Reducer<
    LongWritable, Text, LongWritable, Text> {

    @Override
    protected void reduce(LongWritable key, Iterable<Text>
        values, Context context) throws IOException,
        InterruptedException {
        TreeMap<Double, Long> closestKeys = new TreeMap
            <>();

        Text valueIn;

        while (values.iterator().hasNext()){
            valueIn = values.iterator().next();
            String[] aux = valueIn.toString().split("\t");
            long id = Long.parseLong(aux[0]);
            double distance = Double.parseDouble(aux[1]);
            closestKeys.put(distance, id);
            if (closestKeys.size() > 10){
                closestKeys.remove(closestKeys.lastKey());
            }
        }
        for (Double id: closestKeys.keySet()){
            context.write(key, new Text(closestKeys.get(id)
                + "\t" + id));
        }
    }
}

```

```

    }
}

}

```

B.3 Reducer

```

package reducer;

import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Reducer;

import java.io.IOException;
import java.util.TreeMap;

public class KNNSearchReducer extends Reducer<LongWritable
    , Text, LongWritable, Text> {

    @Override
    protected void reduce(LongWritable key, Iterable<Text>
        values, Context context) throws IOException,
        InterruptedException {
        TreeMap<Double, Long> closestKeys = new TreeMap
            <>();

        Text valueIn;

        while (values.iterator().hasNext()){
            valueIn = values.iterator().next();
            String [] aux = valueIn.toString().split("\\t");

```

```

        long id = Long.parseLong(aux[0]);
        double distance = Double.parseDouble(aux[1]);
        closestKeys.put(distance, id);
        if (closestKeys.size() > 10){
            closestKeys.remove(closestKeys.lastKey());
        }
    }
    String out = "";
    for (Long id: closestKeys.values()){
        out += id + "\t";
    }
    context.write(key, new Text(out));
}
}

```

Apêndice C: Implementação Box-Counting

C.1 Mapper - Job 1

```
package mapper;

import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.NullWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Mapper;

import java.io.IOException;

public class BoxCountingMapperV2 extends Mapper<
    LongWritable, Text, Text, NullWritable>{
    double scale, menor, maior, passo;
    @Override
    protected void map(LongWritable key, Text value,
        Context context) throws IOException,
        InterruptedException {
        String[] aux = value.toString().split("\\t");
        double x = Double.parseDouble(aux[0]);
        double y = Double.parseDouble(aux[1]);

        for (double s = menor; s < maior; s+=passo) {
            scale = s;
            int box_x = (int) Math.floor(x/scale);
```

```

        int box_y = (int) Math.floor(y/scale);
        context.write(new Text(s+"\t"+box_x+" "+box_y)
            , NullWritable.get());
    }

}

@Override
protected void setup(Context context) throws
    IOException, InterruptedException {
    menor = Double.parseDouble(context.
        getConfiguration().get("menor"));
    maior = Double.parseDouble(context.
        getConfiguration().get("maior"));
    passo = Double.parseDouble(context.
        getConfiguration().get("passo"));
}
}

```

C.2 Combiner - Job 1

Nessa implementação o código utilizado como *combiner* foi o mesmo código usado no *reducer*.

C.3 Reducer - Job 1

```

package reducer;

import org.apache.hadoop.io.NullWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Reducer;

```

```

import java.io.IOException;

public class BoxCountingReducerV2 extends Reducer<Text,
    NullWritable, Text, NullWritable>{

    @Override
    protected void reduce(Text key, Iterable<NullWritable>
        values, Context context) throws IOException,
        InterruptedException {
        context.write(key, NullWritable.get());
    }
}

```

C.4 Mapper - Job 2

```

package mapper;

import org.apache.hadoop.io.DoubleWritable;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Mapper;

import java.io.IOException;

public class BoxCountingMapperV2Part2 extends Mapper<
    LongWritable, Text, DoubleWritable, IntWritable>{
    @Override
    protected void map(LongWritable key, Text value,
        Context context) throws IOException,

```

```

        InterruptedException {
            String aux[] = value.toString().split("\t");
            context.write(new DoubleWritable(Double.
                parseDouble(aux[0])), new IntWritable(1));
        }
    }
}

```

C.5 Combiner - Job 2

```

package reducer;

import org.apache.hadoop.io.DoubleWritable;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.mapreduce.Reducer;

import java.io.IOException;

public class BoxCountingCombinerV2Part2 extends Reducer<
    DoubleWritable, IntWritable, DoubleWritable,
    IntWritable>{
    @Override
    protected void reduce(DoubleWritable key, Iterable<
        IntWritable> values, Context context) throws
        IOException, InterruptedException {
        int soma=0;
        while (values.iterator().hasNext()){
            soma += values.iterator().next().get();
        }
        context.write(key, new IntWritable(soma));
    }
}

```

C.6 Reducer - Job 2

```
package reducer;

import org.apache.hadoop.io.DoubleWritable;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.mapreduce.Reducer;

import java.io.IOException;

public class BoxCountingReducerV2Part2 extends Reducer<
    DoubleWritable, IntWritable, DoubleWritable,
    DoubleWritable>{
    @Override
    protected void reduce(DoubleWritable key, Iterable<
        IntWritable> values, Context context) throws
        IOException, InterruptedException {
        int soma = 0;
        while (values.iterator().hasNext()){
            soma += values.iterator().next().get();
        }
        DoubleWritable keyOut = new DoubleWritable(Math.
            log10(1d/key.get()));
        DoubleWritable valueOut = new DoubleWritable(Math.
            log10(soma));
        context.write(keyOut, valueOut);
    }
}
```

Apêndice D: Implementação Dimensão de Correlação

D.1 Mapper

```
package mapper;

import model.MAP_COUNTER;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.DoubleWritable;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Mapper;

import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;
import java.util.ArrayList;
import java.util.List;

public class GPCPMapper extends Mapper<LongWritable, Text,
    DoubleWritable, Text> {

    static List<Double> serie = null;

    @Override
```

```

protected void setup(Context context) throws
    IOException, InterruptedException {
    int qtdDados = Integer.parseInt(context.
        getConfiguration().get("dados"));
    if (serie == null) {
        Path[] localPaths = context.getLocalCacheFiles
            ();
        serie = new ArrayList<>();

        try {
            BufferedReader readBuffer1 = new
                BufferedReader(new FileReader(
                    localPaths[0].getName().trim()));
            String line;
            int qtd = 0;
            while ((line = readBuffer1.readLine()) !=
                null && qtd < qtdDados) {
                serie.add(Double.parseDouble(line));
                qtd++;
            }
            readBuffer1.close();

        } catch (IOException e) {
            throw new RuntimeException(e);
        }

    }
}

@Override

```

```

protected void map(LongWritable key, Text value,
    Context context) throws IOException,
    InterruptedException {
    String aux[] = value.toString().split("\\t");
    int dim = Integer.parseInt(aux[0]);
    double lr = Double.parseDouble(aux[1]);

    int tao = 60;

    double rr;
    double teta, res;
    double soma = 0;

    int indice1, indice2;

    rr = Math.pow(10.0, lr);
    teta = 0.0;

    for (int i = 0; i < (serie.size() - 1 - (dim - 1)
        * tao); i++) {
        if (i%10000==0){
            context.getCounter(MAP_COUNTER.
                CORREL_COUNT_X).increment(1);
            context.progress();
        }

        for (int j = i + 1; j < (serie.size() - (dim -
            1) * tao); j++) {

            soma = 0.0;
            if (i != j) {

```

```

        for (int n = 0; n < dim; n++) {
            indice1 = i + n * tao;
            indice2 = j + n * tao;
            soma += Math.pow((serie.get(
                indice1) - serie.get(indice2)),
                2);
        }
        soma = Math.sqrt(soma);
        if (rr > soma) {
            teta += 2.0;
        }
    }
}

long tam = (serie.size() - (dim - 1) * tao);
res = teta / (tam * (tam - 1));
res = Math.log10(res);

context.write(new DoubleWritable(lr), new Text(dim
    + "\t" + res));

}
}

```

D.2 Reducer

```
package reducer;
```

```

import org.apache.hadoop.io.DoubleWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Reducer;
import writable.IntArrayWritable;

import java.io.IOException;

public class GPCPReducer extends Reducer<DoubleWritable,
    Text, DoubleWritable, Text> {

    @Override
    protected void reduce(DoubleWritable key, Iterable<
        Text> values, Context context) throws IOException,
        InterruptedException {
        double saida[] = new double[9];

        while (values.iterator().hasNext()){
            String aux[] = values.iterator().next().
                toString().split("\t");
            int idx = Integer.parseInt(aux[0]) - 2;
            double value = Double.parseDouble(aux[1]);

            saida[idx] = value;
        }

        String out = "";
        for (int i = 0; i < saida.length; i++) {
            if (i == saida.length - 1){
                out += saida[i] + "";
            } else {
                out += saida[i] + ";";
            }
        }
    }
}

```

```
        }  
    }  
    context.write(key,new Text(out));  
}  
}
```